String Matching with Variable Length Gaps*

Philip Bille Inge Li Gørtz† Hjalte Wedel Vildhøj David Kofoed Wind

Technical University of Denmark, DTU Informatics, March 2012

Abstract

We consider string matching with variable length gaps. Given a string T and a pattern P consisting of strings separated by variable length gaps (arbitrary strings of length in a specified range), the problem is to find all ending positions of substrings in T that match P. This problem is a basic primitive in computational biology applications. Let m and n be the lengths of P and T, respectively, and let k be the number of strings in P. We present a new algorithm achieving time $O(n \log k + m + \alpha)$ and space $O(m + A)$, where A is the sum of the lower bounds of the lengths of the gaps in P and α is the total number of occurrences of the strings in P within T. Compared to the previous results this bound essentially achieves the best known time and space complexities simultaneously. Consequently, our algorithm obtains the best known bounds for almost all combinations of m, n, k, A, and α. Our algorithm is surprisingly simple and straightforward to implement. We also present algorithms for finding and encoding the positions of all strings in P for every match of the pattern.

1 Introduction

Given integers a and b, $0 \leq a \leq b$, a *variable length gap* $g\{a, b\}$ is an arbitrary string, over a finite alphabet Σ, of length between a and b, both inclusive. A *variable length gap pattern* (abbreviated VLG pattern) P is the concatenation of a sequence of strings and variable length gaps, that is, P is of the form

$$P = P_1 \cdot g\{a_1, b_1\} \cdot P_2 \cdot g\{a_2, b_2\} \cdots g\{a_{k-1}, b_{k-1}\} \cdot P_k \ .$$

A VLG pattern P *matches* a substring S of T iff $S = P_1 \cdot G_1 \cdots G_{k-1} \cdot P_k$, where G_i is any string of length between a_i and b_i, $i = 1, \ldots, k-1$. Given a string T and a VLG pattern P, the *variable length gap problem* (VLG problem) is to find all ending positions of substrings in T that match P.

Example 1 *As an example, consider the problem instance over the alphabet* $\Sigma = \{A, G, C, T\}$*:*

$$T = ATCGGCTCCAGACCAGTACCCGTTCCGTGGT$$
$$P = A \cdot g\{6, 7\} \cdot CC \cdot g\{2, 6\} \cdot GT$$

The solution to the problem instance is the set of positions $\{17, 28, 31\}$*. For example the solution contains 17, since the substring ATCGGCTCCAGACCAGT, ending at position 17 in* T*, matches* P*.*

*An extended abstract of this paper appeared in proceedings of the 17th Symposium on String Processing and Information Retrieval.

†Supported by the Danish Council for Independent Research | Natural Sciences

Variable length gaps are frequently used in computational biology applications [7, 8, 14, 16, 17]. For instance, the PROSITE data base [5, 10] supports searching for proteins specified by VLG patterns.

1.1 Previous Work

We briefly review the main worst-case bounds for the VLG problem. As above, let $P = P_1 \cdot g\{a_1, b_1\} \cdot P_2 \cdot g\{a_2, b_2\} \cdots g\{a_{k-1}, b_{k-1}\} \cdot P_k$ be a VLG pattern consisting of k strings, and let T be a string. To state the bounds, let $m = \sum_{i=1}^{k} |P_i|$ be the sum of the lengths of the strings in P and let n be the length of T.

The simplest approach to solve the VLG problem is to translate P into a regular expression and then use an algorithm for regular expression matching. Unfortunately, the translation produces a regular expression significantly longer than P, resulting in an inefficient algorithm. Specifically, suppose that the alphabet Σ contains σ characters, that is, $\Sigma = \{c_1, \ldots, c_\sigma\}$. Using standard regular expression operators (union and concatenation), we can translate $g\{a, b\}$ into the expression

$$g\{a, b\} = \overbrace{C \cdots C}^{a}\overbrace{(C|\epsilon) \cdots (C|\epsilon)}^{b-a},$$

where C is shorthand for the expression $(c_1 \mid c_2 \mid \ldots c_\sigma)$. Hence, a variable length gap $g\{a, b\}$, represented by a constant length expression in P, is translated into a regular expression of length $\Omega(\sigma b)$. Consequently, a regular expression R corresponding to P has length $\Omega(B\sigma + m)$, where $B = \sum_{i=1}^{k-1} b_i$ is the sum of the upper bounds of the gaps in P. Using Thompson's textbook regular expression matching algorithm [20] this leads to an algorithm for the VLG problem using $O(n(B\sigma + m))$ time. Even with the fastest known algorithms for regular expression matching this bound can only be improved by at most a polylogarithmic factor [2, 3, 15, 18].

Several algorithms that improve upon the direct translation to a regular expression matching problem have been proposed [4, 6–8, 12–14, 16, 17, 19]. Some of these are able to solve more general versions of the problem, such as searching for patterns that also contain character classes and variable length gaps with negative length. Most of the algorithms are based on fast simulations of non-deterministic finite automata. In particular, Navarro and Raffinot [17] gave an algorithm using $O(n(\frac{m+B}{w} + 1))$ time, where w is the number of bits in a memory word. Fredrikson and Grabowski [7, 8] improved this bound for the case when all variable length gaps have lower bound 0 and identical upper bound b. Their fastest algorithm achieves $O(n(\frac{m \log\log b}{w} + 1))$ time. Very recently, Bille and Thorup [4] gave an algorithm using $O(n(k\frac{\log w}{w} + \log k) + m \log m + A)$ time and $O(m + A)$ space, where $A = \sum_{i=1}^{k-1} a_i$ is the sum of the lower bounds on the lengths of the gaps. Note that if we assume that the nk term dominates and ignore the $w/\log w$ factor, the time bound reduces to $O(nk)$.

An alternative approach, suggested independently by Morgante et al. [13] and Rahman et al. [19], is to design algorithms that are efficient in terms of the total number of occurrences of the k strings $P_1, \ldots, P_k$ within T. Let α be this number, e.g., in Example 1 A, CC, and GT occur 5, 5, and 4 times in T. Hence, $\alpha = 5 + 5 + 4 = 14$. Rahman et al. [19] gave an algorithm using $O(n \log k + m + \alpha \log(\max_{1 \le i < k}(b_i - a_i)))$ time[1]. Morgante et al. [13] gave a faster algorithm using $O(n \log k + m + \alpha)$ time. Each of the k strings in P can occur at most n times and therefore $\alpha \le nk$.

[1]The bound stated in the paper does not include the $\log k$ factor, since they assume that the size of the alphabet is constant. We make no assumption on the alphabet size and therefore include it here.

Hence, in the typical case when the strings occur less frequently, i.e, $\alpha = o(n(k\frac{\log w}{w} + \log k))$, these approaches are faster. However, unlike the automata based algorithm that only use $O(m + A)$ space, both of these algorithm use $\Theta(m + \alpha)$ space. Since α typically increases with the length of T, the space usage of these algorithms is likely to quickly become a bottleneck for processing large biological data bases.

1.2 Our Results

We address the basic question of whether it is possible to design an algorithm that simultaneously is fast in the total number of occurrences of the k strings and uses little space. We show the following result.

Theorem 1 *Given a string T and a VLG pattern P with k strings, we can solve the variable length gaps matching problem in time $O(n \log k + m + \alpha)$ and space $O(m + A)$. Here, α is the number of occurrences of the strings of P in T and A is the sum of the lower bounds of the gaps.*

Hence, we match the best known time bounds in terms of α and the space for the fastest automata based approach. Consequently, whenever $\alpha = o(n(k\frac{\log w}{w} + \log k))$ the time and space bounds of Theorem 1 are the best known. Our algorithm uses a standard comparison based version of the Aho-Corasick automaton for multi-string matching [1]. If the size of the alphabet is constant or we use hashing the $\log k$ factor in the running time disappears. Furthermore, our algorithm is surprisingly simple and straightforward to implement.

In some cases, we may also be interested in outputting not only the ending positions of matches of P, but also the positions of the individual strings in P for each match of P in T. Note that there can be exponentially many, i.e., $\Omega(\prod_{i=1}^{k-1}(1+b_i-a_i))$, of these occurrences ending at the same position in T. Morgante et al. [13] showed how to encode all of these in a graph of size $\Theta(\alpha)$. We show how our algorithm can be extended to efficiently output such a graph. Furthermore, we show two solutions for outputting the positions encoded in the graph. Both solutions use little space since they avoid the need to store the entire graph. The first solution is a black-box solution that works with any algorithm for constructing the graph. The second is a direct approach obtained using a simple extension of our algorithm.

Recently, Haapasalo et al. [9] studied practical algorithms for an extension of the VLG problem that allows multiple patterns and gaps with unbounded upper bounds. We note that the result of Theorem 1 is straightforward to generalize to this case.

1.3 Technical Overview

The previous works by Morgante et al. [13] and Rahman et al. [19] find all of the α occurrences of the strings $P_1, \ldots, P_k$ of P in T using a standard multi-string matching algorithm (see Section 2.1). From these, they construct a graph of size $\Omega(\alpha)$ to represent possible combinations of string occurrences that can be combined to form occurrences of P.

Our algorithm similarly finds all of the occurrences of the strings of P in T. However, we show how to avoid constructing a large graph representing the possible combinations of occurrences. Instead we present a way to efficiently represent sufficient information to correctly find the occurrences of P, leading to a significant space improvement from $O(m + \alpha)$ to $O(m + A)$. Surprisingly, the algorithm needed to achieve this space bound is very simple, and only requires maintaining a set

of sorted lists of disjoint intervals. Even though the algorithm is simple the space bound achieved by it is non-obvious. We give a careful analysis leading to the $O(m + A)$ space bound.

Our space-efficient black-box solution for reporting the positions of the individual strings in P for each match of P in T is obtained by constructing the graph for overlapping chunks of T of size $2(m + B)$. Hence the solution is parametrized by the time and space complexity of the actual algorithm used to construct the graph.

2 Algorithm

In this section we present the algorithm. For completeness, we first briefly review the classical Aho-Corasick algorithm for multiple string matching in Section 2.1. We then define the central idea of *relevant occurrences* in Section 2.2. We present the full algorithm in Section 2.3 and analyze it in Section 3.

2.1 Multi-String Matching

Given a set of pattern strings $\mathcal{P} = \{P_1, \ldots, P_k\}$ of total length m and a text T of length n the *multi-string matching problem* is to report all occurrences of each pattern string in T. Aho and Corasick [1] generalized the classical Knuth-Morris-Pratt algorithm [11] for single string matching to multiple strings. The *Aho-Corasick automaton* (AC-automaton) for $\mathcal{P}$, denoted $\text{AC}(\mathcal{P})$, consists of the trie of the patterns in $\mathcal{P}$. Hence, any path from the root of the trie to a state s corresponds to a prefix of a pattern in $\mathcal{P}$. We denote this prefix by $\text{path}(s)$. For each state s there is also a special *failure transition* pointing to the unique state s' such that $\text{path}(s')$ is the longest prefix of a pattern in $\mathcal{P}$ matching a proper suffix of $\text{path}(s)$. Note that the depth of s' in the trie is always strictly smaller for non-root states than the depth of s.

Finally, for each state s we store the subset $\text{occ}(s) \subseteq \mathcal{P}$ of patterns that match a suffix of $\text{path}(s)$. Since the patterns in $\text{occ}(s)$ share suffixes we can represent $\text{occ}(s)$ compactly by storing for s the index of the longest string in $\text{occ}(s)$ and a pointer to the state s' such that $\text{path}(s')$ is the second longest string if any. In this way we can report $\text{occ}(s)$ in $O(|\text{occ}(s)|)$ time.

The maximum outdegree of any state is bounded by the number of leaves in the trie which is at most k. Hence, using a standard comparison-based balanced search tree to index the trie transitions out of each state we can construct $\text{AC}(\mathcal{P})$ in $O(m \log k)$ time and $O(m)$ space.

To find the occurrences of $\mathcal{P}$ in T, we read the characters of T from left-to-right while traversing $\text{AC}(\mathcal{P})$ to maintain the longest prefix of the strings in $\mathcal{P}$ matching T. At a state s and character c we proceed as follows. If c matches the label of a trie transition t from s, the next state is the child endpoint of t. Otherwise, we recursively follow failure transitions from s until we find a state s' with a trie transition t' labeled c. The next state is then the child endpoint of t'. If no such state exists, the next state is the root of the trie. For each failure transition traversed in the algorithm we must traverse at least as many trie transitions. Therefore, the total time to traverse $\text{AC}(\mathcal{P})$ and report occurrences is $O(n \log k + \alpha)$, where α is the total number of occurrences.

Hence, the Aho-Corasick algorithm solves multi-string matching in $O((n + m) \log k + \alpha)$ time and $O(m)$ space.

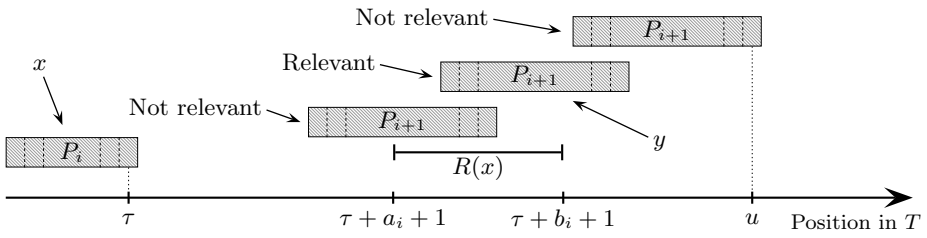

Figure 1: In this figure x is an occurrence of P_i in T reported at position τ. The first and last occurrence of P_{i+1} start outside $R(x)$ thereby violating the ith gap constraint, so these occurrences are not relevant compared to x. The second occurrence y of P_{i+1} starts in $R(x)$, so if x is itself relevant, then y is also relevant.

2.2 Relevant Occurrences

For a substring x of T, let *startpos*(x) and *endpos*(x) denote the start and end position of x in T, respectively. Let x be an occurrence of P_i with $\tau =$ *endpos*(x) in T, and let $R(x)$ denote the range $[\tau + a_i + 1; \tau + b_i + 1]$ in T. An occurrence y of P_i in T is a *relevant occurrence* of P_i iff $i = 1$ or *startpos*$(y) \in R(x)$, for some relevant occurrence x of P_{i-1}. See Figure 1 for an example. Relevant occurrences are similar to the *valid occurrences* defined in [19]. The difference is that a valid occurrence is an occurrence of P_{i+1} that is in $R(x)$ for *any* occurrence x of P_i in T, i.e., x need not be a valid occurrence itself.

From the definition of relevant occurrences, it follows directly that we can solve the VLG problem by finding the relevant occurrences of P_k in T. Specifically, we have the following result.

Lemma 1 *Let $x_1, \ldots, x_k$ be occurrences in T of the subpatterns $P_1, \ldots, P_k$ from the VLG pattern $P = P_1 \cdot g\{a_1, b_1\} \cdot P_2 \cdot g\{a_2, b_2\} \cdots P_k$. Together, these k occurrences constitute an occurrence of P in T if and only if startpos$(x_{i+1}) \in R(x_i)$ for all $i = 1, \ldots, k-1$.*

2.3 The Algorithm

Algorithm 1 computes the relevant occurrences of P_k using the output from the AC automaton. The idea behind the algorithm is to keep track of the ranges defined by the relevant occurrences of each subpattern P_i, such that we efficiently can check if an occurrence of P_i is relevant or not. More precisely, for each subpattern P_i, $i = 2, \ldots, k$, we maintain a sorted list L_i containing the ranges defined by previously reported relevant occurrences of P_{i-1}. When an occurrence of P_i is reported by the AC automaton, we can determine whether it is relevant by checking if it starts in a range contained in L_i (step 2b). Initially, the lists $L_2, L_3, \ldots, L_k$ are empty. When a relevant occurrence of P_i is reported, we add the range defined by this new occurrence to the end of L_{i+1}. In case the new range $[s, t]$ overlaps or adjoins the last range $[q, r]$ in L_{i+1} ($s \leq r + 1$) we merge the two ranges into a single range $[q, t]$.

Let τ denote the current position in T. A range $[a, b] \in L_i$ is *dead at position* τ iff $b < \tau - |P_i|$. When a range is dead no future occurrences y of P_i can start in that range since *endpos*$(y) \geq \tau$ implies *startpos*$(y) \geq \tau - |P_i|$. In Figure 1 the range $R(x)$ defined by x dies when position u is reached. Our algorithm repeatedly removes any dead ranges to limit the size of the lists $L_2, L_3, \ldots, L_k$. To remove the dead ranges in step 2a we traverse the list and delete all dead ranges until we meet a

Algorithm 1 Algorithm solving the VLG problem for a VLG pattern P and a string T.

1. Build the AC-automaton for the subpatterns $P_1, P_2, \ldots, P_k$.
2. Process T using the automaton and each time an occurrence x of P_i is reported at position $\tau = \mathit{endpos}(x)$ in T do:
   (a) Remove any dead ranges from the lists L_i and L_{i+1}.
   (b) If $i = 1$ or $\tau - |P_i| = \mathit{startpos}(x)$ is contained in the first range in L_i do:
      i. If $i < k$: Append the range $R(x) = [\tau + a_i + 1; \tau + b_i + 1]$ to the end of L_{i+1}. If the range overlaps or adjoins the last range in L_{i+1}, the two ranges are merged into a single range.
      ii. If $i = k$: Report τ.

range that is not dead. Since the lists are sorted, all remaining ranges in the list are still alive. See Figure 2 for an example.

3 Analysis

We now show that Algorithm 1 solves the VLG problem in time $O(n \log k + m + \alpha)$ and space $O(m + A)$, implying Theorem 1.

3.1 Correctness

To show that Algorithm 1 finds exactly the relevant occurrences of P_k, we show by induction on i that the algorithm in step 2b correctly determines the relevancy of all occurrences of P_i, $i = 1, 2, \ldots, k$, in T.

Base case: All occurrences of P_1 are by definition relevant and Algorithm 1 correctly determines this in step 2b.

Inductive step: Let y be an occurrence of P_i, $i > 1$, that is reported at position τ. There are two cases to consider.

1. y is relevant. By definition there is a relevant occurrence x of P_{i-1} in T, such that $\mathit{startpos}(y) = \tau - |P_i| \in R(x)$. By the induction hypothesis x was correctly determined to be relevant by the algorithm. Since $\mathit{endpos}(x) < \tau$, $R(x)$ was appended to L_i earlier in the execution of the algorithm. It remains to show that the range containing $\mathit{startpos}(y)$ is the first range in L_i in step 2b. When removing the dead ranges in L_i in step 2a, all ranges $[a, b]$ where $b < \tau - |P_i|$ are removed. Therefore the range containing $\tau - |P_i| = \mathit{startpos}(y)$ is the first range in L_i after step 2a. It follows that the algorithm correctly determines that y is relevant.
2. y is not relevant. Then there exists no relevant occurrence x of P_{i-1} such that $\mathit{startpos}(y) \in R(x)$. By the induction hypothesis there is no range in L_i containing $\mathit{startpos}(y)$, since the algorithm only append ranges when a relevant occurrence is found. Consequently, the algorithm correctly determines that y is not relevant.

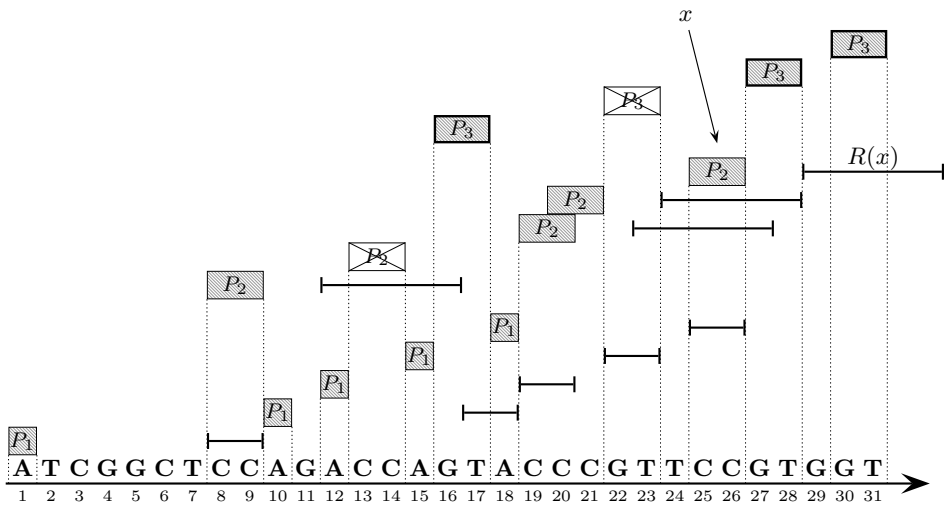

Figure 2: Illustrating how Algorithm 1 finds the occurrences of the VLG pattern $P = \mathrm{A} \cdot g\{6,7\} \cdot \mathrm{CC} \cdot g\{2,6\} \cdot \mathrm{GT}$ in the text T from Example 1. The figure shows the occurrences of the subpatterns $P_1 = \mathrm{A}$, $P_2 = \mathrm{CC}$ and $P_3 = \mathrm{GT}$ and the ranges they define in the text T. Occurrences which are not relevant are crossed out. The bold occurrences of P_3 are the relevant occurrences of P_k and their end positions 17,28 and 31 constitute the solution to the VLG problem. Consider the point in the execution of the algorithm when the occurrence x of P_2 at position $\tau = 26$ is reported by the Aho-Corasick automaton. At this time $L_2 = \big[\ [17;20], [22;23], [25;26]\ \big]$ and $L_3 = \big[\ [23,28]\ \big]$. The ranges $[17;20]$ and $[22;23]$ are now dead and are removed from L_2 in step 2a. In step 2b the algorithm determines that x is relevant and $R(x) = [29;33]$ is appended to L_3: $L_3 = \big[\ [23;33]\ \big]$.

3.2 Time and Space Complexity

The AC automaton for the subpatterns $P_1, P_2, \ldots, P_k$ can be built in time $O(m \log k)$ using $O(m)$ space, where $m = \sum_{i=1}^{k} |P_i|$. In the trivial case when $m > n$ we do not need to build the automaton. Hence, we will assume that $m \leq n$ in the following analysis. For each of the α occurrences of the strings $P_1, P_2, \ldots, P_k$ Algorithm 1 first removes the dead ranges from L_i and L_{i+1} and performs a number of constant-time operations. Since both lists are sorted, the dead ranges can be removed by traversing the lists from the beginning. At most α ranges are ever added to the lists, and therefore the algorithm spends $O(\alpha)$ time in total on removing dead ranges. The total time is therefore $O((n+m)\log k + \alpha) = O(n \log k + m + \alpha)$.

To prove the space bound, we first show the following lemma.

Lemma 2 *At any time during the execution of the algorithm we have*

$$|L_i| \leq \left\lfloor \frac{2c_{i-1} + |P_i| + a_{i-1}}{c_{i-1} + 1} \right\rfloor = O\left(\frac{|P_i| + a_{i-1}}{b_{i-1} - a_{i-1} + 2} \right),$$

for $i = 2, 3, \ldots, k$, where $c_i = b_i - a_i + 1$.

Proof. Consider list L_i for some $i = 2, \ldots, k$. Referring to Algorithm 1, the size of the list L_i is only increased in step 2(b)i, when a range $R(x_j)$ defined by a relevant occurrence x_j of P_{i-1} is reported and $R(x_j)$ *does not* adjoin or overlap the last range in L_i.

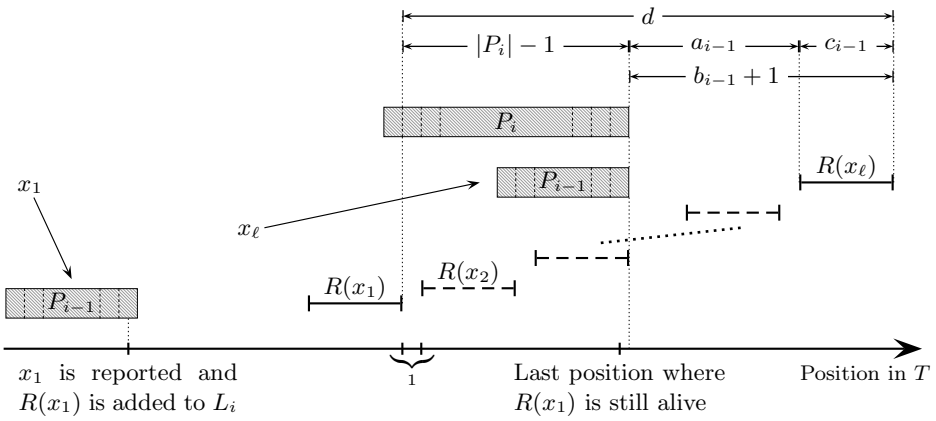

Figure 3: The worst-case situation where ℓ, the maximum number of ranges are present in L_i. The figure only shows the first and the last occurrence of P_{i-1} (x_1 and x_ℓ) defining the ℓ ranges.

Let $R(x_1) = [s, t]$ be the first range in L_i at an arbitrary time in the execution of the algorithm. We bound the number of additional ranges that can be added to L_i from the time $R(x_1)$ became the first range in L_i until $R(x_1)$ is removed. The last position where $R(x_1)$ is still alive is $\tau_a = t+|P_i|-1$. If a relevant occurrence x_ℓ of P_{i-1} ends at this position, then the range $R(x_\ell) = [\tau_a + a_{i-1} + 1; \tau_a + b_{i-1} + 1]$ is appended to L_i. Hence, the maximum number of positions d from t to the end of $R(x_\ell)$ is

$$\begin{aligned} d &= \tau_a + b_{i-1} + 1 - t \\ &= (t + |P_i| - 1) + b_{i-1} + 1 - t \\ &= |P_i| + b_{i-1} \\ &= |P_i| + a_{i-1} + c_{i-1} - 1 \ . \end{aligned}$$

In the worst case, all the ranges in L_i are separated by exactly one position as illustrated in Figure 3. Therefore at most $\lfloor d/(c_{i-1}+1) \rfloor$ additional ranges can be added to L_i before $R(x_1)$ is removed. Counting in $R(x_1)$ yields the following bound on the size of L_i

$$|L_i| \leq \left\lfloor \frac{d}{c_{i-1}+1} \right\rfloor + 1 = \left\lfloor \frac{2c_{i-1} + |P_i| + a_{i-1}}{c_{i-1}+1} \right\rfloor = O\left(\frac{|P_i| + a_{i-1}}{b_{i-1} - a_{i-1} + 2} \right) \ .$$

□

By Lemma 2 the total number of ranges stored at any time during the processing of T is at most

$$O\left(\sum_{i=2}^{k} \frac{|P_i| + a_{i-1}}{b_{i-1} - a_{i-1} + 2} \right) = O\left(\sum_{i=1}^{k-1} \frac{|P_{i+1}|}{b_i - a_i + 2} + \sum_{i=1}^{k-1} \frac{a_i}{b_i - a_i + 2} \right) = O\left(m + A\right) \ .$$

Each range can be stored using $O(1)$ space, so this is an upper bound on the space needed to store the lists $L_2, \ldots, L_k$. The AC-automaton uses $O(m)$ space, so the total space required by our algorithm is $O(m + A)$.

In summary, the algorithm uses $O(n \log k + m + \alpha)$ time and $O(m + A)$ space. This completes the proof of Theorem 1.

4 Complete Characterization of Occurrences

In this section we show how our algorithm can be extended to report not only the end position of P_k, but also the positions of $P_1, P_2, \ldots, P_{k-1}$ for each occurrence of P in T.

The main idea is to construct a graph that encodes all occurrences of the VLG-pattern using $O(\alpha)$ space. For each occurrence of the VLG-pattern, the positions of the individual subpatterns can be reported by traversing this graph. This approach was also used by Rahman et al. [19] and Morgante et al. [13]. We give a fast new algorithm for constructing this graph and show a black-box solution that can report the occurrences of the VLG-pattern without storing the complete graph.

We introduce the following simple definitions. If P occurs in the text T, then a *match combination* is a sequence $e_1, \ldots, e_k$ of end positions of $P_1, \ldots, P_k$ in T corresponding to the match. The total number of match combinations of P in T is denoted β. Note that there can be many match combinations corresponding to a single match. See Figure 4.

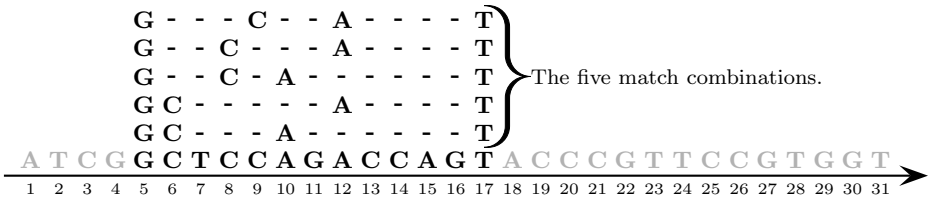

Figure 4: The text sequence is the same as in the previous examples. The substring S from position 5 to 17 (highlighted in bold) matches the VLG-pattern $Q = G \cdot g\{0,3\} \cdot C \cdot g\{1,6\} \cdot A \cdot g\{2,7\} \cdot T$. As the figure shows, this match contains the following five match combinations: [5,9,12,17],[5,8,12,17],[5,8,10,17],[5,6,12,17],[5,6,10,17].

Due to the inequality of arithmetic and geometric means, the total number of match combinations β is maximized when the α occurrences are distributed evenly over $P_1, P_2, \ldots, P_k$ and each occurrence of P_i is compatible to all occurrences of P_{i-1} for $i = 2, \ldots, k$. So in the worst case $\beta = \Theta\left((\frac{\alpha}{k})^k\right)$, which is exponential in the number of gaps. All these match combinations can be encoded in a directed graph using $O(\frac{\alpha^2}{k})$ space as follows. The nodes in the graph are the relevant occurrences of $P_1, P_2, \ldots, P_k$ in T. Two nodes x of P_{i-1} and y of P_i are connected by an edge from y to x if and only if $startpos(y) \in R(x)$. In that case we also say that x and y are *compatible*. We denote this graph as the *gap graph* for P and T. See Figure 5. Since the number of nodes in the gap graph is at most α, and there are $O\left((\frac{\alpha}{k})^2\right)$ edges between the k layers in the worst case, we can store the graph using $O(\frac{\alpha^2}{k})$ space.

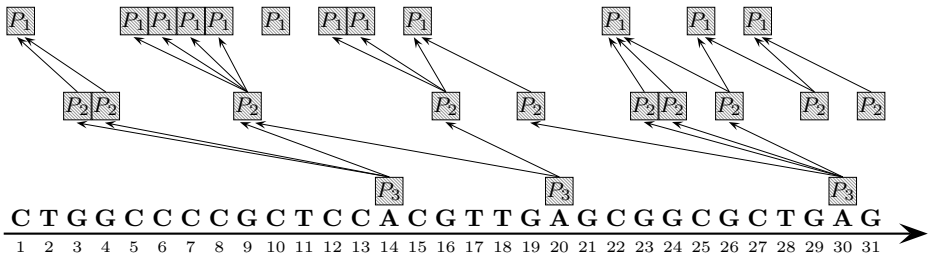

Figure 5: The gap graph for the VLG-pattern $R = C \cdot g\{0,3\} \cdot G \cdot g\{3,10\} \cdot A$ and the text $T =$ CTGGCCCCGCTCCACGTTGAGCGGCGCTGAG.

If the j occurrences $x_1, x_2, \ldots, x_j$ of P_i (appearing in that order in T) are all compatible with the same occurrence y of P_{i+1}, then the j edges $(y, x_1), (y, x_2), \ldots, (y, x_j)$ are all present in the gap graph. Due to the following lemma, the edges $(y, x_2), \ldots, (y, x_{j-1})$ are redundant.

Lemma 3 *Let x_1 and x_2 be two occurrences of P_i, $i = 1, \ldots, k-1$, both compatible with the same occurrence y of P_{i+1}. Assume without loss of generality that $startpos(x_1) < startpos(x_2)$ and let x' be another occurrence of P_i such that $startpos(x_1) \leq startpos(x') \leq startpos(x_2)$, then x' is also compatible with y.*

Proof. Since $startpos(y) \in R(x_1)$ and $startpos(y) \in R(x_2)$, we have that $startpos(y) \in R(x_1) \cap R(x_2)$. Furthermore since $startpos(x_1) \leq startpos(x') \leq startpos(x_2)$, it holds that $R(x_1) \cap R(x_2) \subseteq R(x')$, so $startpos(y) \in R(x')$. □

Leaving out the redundant edges in the gap graph, we get a new graph, which we denote the *implicit gap graph*. For an example, see Figure 6. In this graph the out-degree of each node is at most two, so the number of edges is now linear in the number of nodes, and consequently we can store the implicit gap graph using $O(\alpha)$ space.

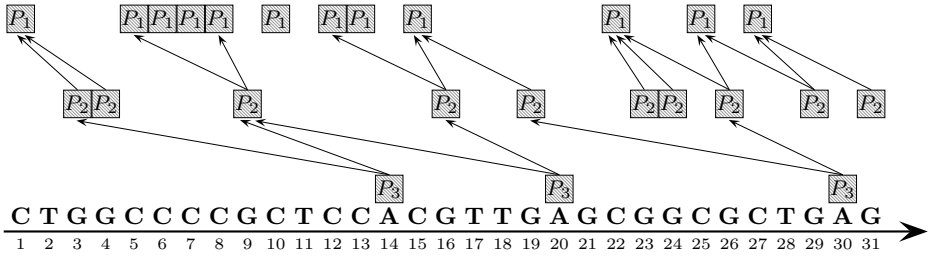

Figure 6: The implicit gap graph for the VLG-pattern $R = C \cdot g\{0,3\} \cdot G \cdot g\{3,10\} \cdot A$ and the text $T =$ CTGGCCCCGCTCCACGTTGAGCGGCGCTGAG. The out-degree of each node is at most two. Compare to Figure 5.

In the context of these new definitions, we are interested in solving the two following problems:

The *reporting variable length gaps problem* (RVLG problem) is to output all match combinations of P in T.

The *implicit reporting variable length gaps problem* (IRVLG problem) is to output the implicit gap graph of all match combinations of P in T.

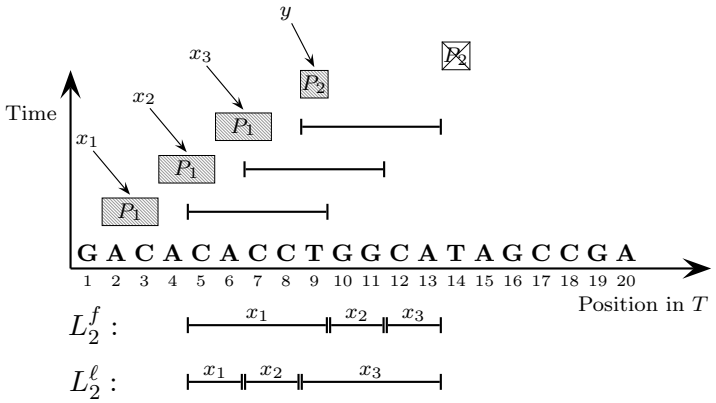

Figure 7: Example showing how the two lists L_2^f and L_2^ℓ store the first and most recent range to cover a position in the text, respectively. The VLG-pattern is $\mathrm{AC} \cdot g\{1,5\} \cdot \mathrm{T}$. When the occurrence y of $P_2 = \mathrm{T}$ at position 9 is reported, we can check the two lists to see that x_1 is the first and x_3 is the last occurrence of P_1 compatible with y.

4.1 Constructing the Implicit Gap Graph

Algorithm 2 describes how to build the implicit gap graph. Recall that in Algorithm 1 the ranges in L_i allowed us to determine the relevancy of a newly reported occurrence x of P_i by inspecting the first range in L_i (after the dead ranges had been removed). To build the implicit gap graph, we need to not only determine the relevancy of x, but also the first and last occurrence of P_{i-1} compatible with x. This information allows us to add the correct edges to the implicit gap graph.

To do this, we replace the list L_i with two lists L_i^f and L_i^ℓ, for $i = 2, \ldots, k$. The idea is that when a position in the text is covered by multiple ranges, L_i^f contains the first range and L_i^ℓ contains the most recent range to cover that position. See Figure 7. Each range $[s,t]$ in L_i^f or L_i^ℓ now also has a reference to the occurrence x of P_{i-1} that defined it, and we will denote the range $[s,t]_x$ to indicate this. When an occurrence x of P_i is reported, we first remove dead ranges from the lists L_i^f, L_i^ℓ, L_{i+1}^f and L_{i+1}^ℓ as was done in Algorithm 1. If x is relevant a node representing x is added to the implicit gap graph in step 2(b)i. In step 2(b)ii, provided that x is not an occurrence of P_1, the two out-going edges of x are added by inspecting L_i^f and L_i^ℓ to determine the first and last occurrence of P_{i-1} compatible with x. Unless x is an occurrence of P_k, the range $R(x) = [\tau + a_i + 1; \tau + b_i + 1]$ is added to the lists L_{i+1}^f and L_{i+1}^ℓ in step 2(b)iii as described in the following section.

4.1.1 Maintaining the Range Lists

When adding a range $[s,t]_x$ defined by an occurrence x of P_i to L_{i+1}^f and L_{i+1}^ℓ, we simply append it to the end of the list if it does not overlap the last range in the list. Otherwise, to avoid overlapping ranges, we appropriately shorten either the newly added range $[s,t]_x$ (for L_{i+1}^f) or the last range in the list (for L_{i+1}^ℓ). The way L_i^f is maintained ensures that the first range that covers some position τ in T will remain the only range covering this position in L_i^f. Conversely, L_i^ℓ will store the most recent range covering τ. In Algorithm 2 the steps 2(b)iii, A, B and C append and possibly shorten the ranges according to this strategy.

Algorithm 2 Algorithm solving the IRVLG problem for a VLG pattern P and a string T.

1. Build the AC-automaton for the subpatterns $P_1, P_2, \ldots, P_k$.
2. Process T using the automaton and each time an occurrence x of P_i is reported at position $\tau = \mathit{endpos}(x)$ in T do:
   (a) Remove any dead ranges from the lists L_i^f, L_i^ℓ, L_{i+1}^f and L_{i+1}^ℓ.
   (b) If $i = 1$ or if $\tau - |P_i| = \mathit{startpos}(x)$ is contained in the first range in L_i^f (i.e., x is a relevant occurrence) do:
      i. Add the node x to the implicit gap graph.
      ii. If $i > 1$: Add the edges (x, y) and (x, z) to the implicit gap graph, where y and z are the occurrences of P_{i-1} defining the first range in L_i^f and L_i^ℓ, respectively.
      iii. If $i < k$: Let $[q, r]_w$ and $[q', r']_{w'}$ denote the first and last range in L_{i+1}^f and L_{i+1}^ℓ, respectively.
         A. Append the range $[\max(r + 1, \tau + a_i + 1), \tau + b_i + 1]_x$ to the end of L_{i+1}^f.
         B. Change the last range in L_{i+1}^ℓ to $[q', \min(r', \tau + a_i)]_{w'}$.
         C. Append the range $[\tau + a_i + 1; \tau + b_i + 1]_x$ to the end of L_{i+1}^ℓ.

4.1.2 Time and Space Analysis

As for Algorithm 1, the time spent for each of the at most α relevant occurrences reported by the AC automaton is amortized constant. Hence the implicit gap graph can be built in $O(n \log k + m + \alpha)$ time. Storing the implicit gap graph for the entire text takes space $O(\alpha)$, since each of the at most α nodes has at most two out-going edges.

We now consider the space needed to store the lists L_i^f and L_i^ℓ. The ranges in L_i^f and L_i^ℓ are no longer guaranteed to have size $c_{i-1} = b_{i-1} - a_{i-1} + 1$ nor being separated by at least one position, so the bound of Lemma 2 needs to be revised, resulting in a slightly increased space bound for storing the lists. Referring to Figure 3, the number of ranges in L_i^f or L_i^ℓ at any point in time is at most

$$d + 1 = c_{i-1} + |P_i| - 1 + a_{i-1} + 1 = |P_i| + b_{i-1} + 1 \; .$$

Summing up, the total space required to store the lists increases from $O(m + A)$ to $O(m + B)$, where $B = \sum_{i=1}^{k-1} b_i$ is the sum of the upper bounds of the lengths of the gaps.

Recapitulating, we have the following theorem

Theorem 2 *The IRVLG problem can be solved in time $O(n \log k + m + \alpha)$ and space $O(m + B + \alpha)$.*

4.2 A Black-Box Solution for Reporting Match Combinations

The number of match combinations, β, can be exponential in the number of gaps. The implicit gap graph space efficiently encodes all of these match combinations in a graph of size $O(\alpha)$. Thus, a straightforward solution to the RVLG problem is to construct the implicit gap graph and subsequently traverse it to report the match combinations. Each of the β match combinations is a sequence of k integers, so this solution to the RVLG takes time $O(n \log k + m + \alpha + k\beta)$ and space $O(m + B + \alpha)$.

We now show that the RVLG problem can be space efficiently solved using any black-box algorithm for the IRVLG problem. The main idea is a simple splitting of T into overlapping smaller substrings of suitable size. We solve the problem for each substring individually and combine the solutions to solve the full problem. By carefully organizing the computation we can efficiently reuse the space needed for the subproblems.

Let $\mathcal{A}_I$ be any algorithm that solves the IRVLG problem in time $t(n, m, k, \alpha)$ and space $s(n, m, k)$, where n, m, k, and α, are the parameters of the input as above. We build a new algorithm $\mathcal{A}_R$ from $\mathcal{A}_I$ that solves the RVLG problem as follows. Assume without loss of generality that n is a multiple of $2(m + B)$. Divide T into $z = \frac{n}{m+B} - 1$ substrings $C_1, \ldots, C_z$, called *chunks*. Each chunk has length $2(m + B)$ and overlaps in $m + B$ characters with each neighbor. We run $\mathcal{A}_I$ on each chunk $C_1, \ldots, C_z$ in sequence to compute the implicit gap graph for each chunk. By traversing the implicit gap graph for each chunk we output the union of the corresponding match combinations. Since each match combination of P in T occurs in at most two neighboring chunks it suffices to only store the implicit gap graph for two chunks at any time.

Next we consider the complexity $\mathcal{A}_R$. Let α_i denote the number of occurrences of the strings of P in C_i. For each chunk we run $\mathcal{A}_I$ to produce the implicit gap graph. Given these we compute the union of match combinations in $O(k\beta)$ time. Hence, algorithm $\mathcal{A}_R$ uses time

$$O\left(\sum_{i=1}^{z} t(2(m + B), m, k, \alpha_i) + k\beta\right).$$

Next consider the space. We only need to store the implicit gap graphs for two chunks at any time. Since the space required for each chunk is $O((m + B)k)$, the total space becomes

$$O\left((m + B)k + s(2(m + B), m, k)\right).$$

The black-box algorithm efficiently converts algorithms for the IRVLG problem to the RVLG problem, resulting in the following theorem.

Theorem 3 *Given an algorithm solving the IRVLG problem in time $t(n, m, k, \alpha)$ and space $s(n, m, k)$, there is an algorithm solving the RVLG problem in time $O\left(\sum_{i=1}^{z} t(2(m + B), m, k, \alpha_i) + k\beta\right)$ and space $O\left((m + B)k + s(2(m + B), m, k)\right)$.*

If we use the result from Theorem 2, we obtain an algorithm that uses time

$$O\left(\left(\sum_{i=1}^{z} t(2(m + B), m, k, \alpha_i)\right) + k\beta\right) =$$
$$O\left(\frac{n}{m + B}\left(2(m + B)\log k + m\right) + \alpha + k\beta\right) = O(n \log k + m + \alpha + k\beta),$$

where the term m in the last expression is needed for the case where $m > n$. The space usage is

$$O\left((m + B)k + s(2(m + B), m, k)\right) = O\left((m + B)k + m + B + \max_{i=1,\ldots,z} \alpha_i\right) = O((m + B)k),$$

where the last equality holds, since $a_i \leq (m + B)k$ for all i. In summary, we have the following result for the RVLG problem.

Theorem 4 *The RVLG problem can be solved in time $O(n \log k + m + \alpha + k\beta)$ and space $O((m + B)k)$.*

4.3 Reporting Match Combinations On the Fly

We now show how a simple extension of our algorithm provides an alternative solution to the RVLG problem achieving the same space and time complexity as the black-box solution. The idea is to use Algorithm 2 and report the match combinations on the fly, while continually removing nodes from the implicit gap graph that no longer can be part of a match combination. We remove the nodes using a method similar to that for removing dead ranges in the lists L_i^f and L_i^ℓ.

We say that a node x of P_i in the implicit gap graph is *dead* if x can not be part of a future match combination. This happens when

$$\tau > \mathit{endpos}(x) + \sum_{j=i+1}^{k} b_{j-1} + |P_j| \; .$$

Like dead ranges, we can remove dead nodes from the implicit gap graph in amortized constant time. Consequently, all match combinations can be reported in time $O(n \log k + m + \alpha + k\beta)$. Removing the dead nodes ensures that the number of P_i nodes in the implicit gap graph at any time is at most $1 + \sum_{j=i+1}^{k} b_{j-1} + |P_j|$. Thus, the total number of nodes never exceeds

$$\sum_{i=1}^{k} 1 + \sum_{j=i+1}^{k} b_{j-1} + |P_j| \;=\; O((m+B)k) \; .$$

In summary, the algorithm solves the RVLG problem in time $O(n \log k + m + \alpha + k\beta)$ and space $O((m+B)k)$, so it provides and alternative proof of Theorem 4.

References

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[2] P. Bille. New algorithms for regular expression matching. In *Proc. 33rd ICALP*, pages 643–654, 2006.

[3] P. Bille and M. Thorup. Faster regular expression matching. In *Proc. 36th ICALP*, pages 171–182, 2009.

[4] P. Bille and M. Thorup. Regular expression matching with multi-strings and intervals. In *Proc. 21st SODA*, 2010.

[5] P. Bucher and A. Bairoch. A generalized profile syntax for biomolecular sequence motifs and its function in automatic sequence interpretation. In *Proc. 2nd ISMB*, pages 53–61, 1994.

[6] M. Crochemore, C. Iliopoulos, C. Makris, W. Rytter, A. Tsakalidis, and K. Tsichlas. Approximate string matching with gaps. *Nordic J. of Computing*, 9(1):54–65, 2002.

[7] K. Fredriksson and S. Grabowski. Efficient algorithms for pattern matching with general gaps, character classes, and transposition invariance. *Inf. Retr.*, 11(4):335–357, 2008.

[8] K. Fredriksson and S. Grabowski. Nested counters in bit-parallel string matching. In *Proc. 3rd LATA*, pages 338–349, 2009.

[9] T. Haapasalo, P. Silvasti, S. Sippu, and E. Soisalon-Soininen. Online dictionary matching with variable-length gaps. In *Proc. 10th SEA*, pages 76–87, 2011.

[10] K. Hofmann, P. Bucher, L. Falquet, and A. Bairoch. The prosite database, its status in 1999. *Nucleic Acids Res*, (27):215–219, 1999.

[11] D. E. Knuth, J. James H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[12] I. Lee, A. Apostolico, C. S. Iliopoulos, and K. Park. Finding approximate occurrences of a pattern that contains gaps. In *Proc. 14th AWOCA*, pages 89–100, 2003.

[13] M. Morgante, A. Policriti, N. Vitacolonna, and A. Zuccolo. Structured motifs search. *J. Comput. Bio.*, 12(8):1065–1082, 2005.

[14] E. W. Myers. Approximate matching of network expressions with spacers. *J. Comput. Bio.*, 3(1):33–51, 1992.

[15] E. W. Myers. A four-russian algorithm for regular expression pattern matching. *J. ACM*, 39(2):430–448, 1992.

[16] G. Myers and G. Mehldau. A system for pattern matching applications on biosequences. *CABIOS*, 9(3):299–314, 1993.

[17] G. Navarro and M. Raffinot. Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching. *J. Comput. Bio.*, 10(6):903–923, 2003.

[18] G. Navarro and M. Raffinot. New techniques for regular expression searching. *Algorithmica*, 41(2):89–116, 2004.

[19] M. S. Rahman, C. S. Iliopoulos, I. Lee, M. Mohamed, and W. F. Smyth. Finding patterns with variable length gaps or don't cares. In *Proc. 12th COCOON*, pages 146–155, 2006.

[20] K. Thompson. Regular expression search algorithm. *Commun. ACM*, 11:419–422, 1968.