String Matching with Variable Length Gaps

Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and David Kofoed Wind

Technical University of Denmark

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 $O((n+m)\log k+\alpha)$ 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.

1 Introduction

Given integers a and b, $0 \leq a \leq b$, a *variable length gap* $g\{a, b\}$ is an arbitrary string over 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 $\Sigma = \{A, G, C, T\}$:

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

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.

Variable length gaps are frequently used in computational biology applications [15, 13, 16, 7, 8]. For instance, the PROSITE data base [5, 9] 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 [19] 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 [14, 17, 2, 3].

Several algorithms that improve upon the direct translation to a regular expression matching problem have been proposed [15, 13, 6, 16, 11, 12, 18, 7, 8, 4]. 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 [16] 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. [12] and Rahman et al. [18], 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. [18] gave an algorithm using

$O((n+m)\log k + \alpha \log(\max_{1 \leq i < k}(b_i - a_i)))$ time[1]. Morgante et al. [12] gave a faster algorithm using $O((n+m)\log k + \alpha)$ time. Each of the k strings in P can occur at most n times and therefore $\alpha \leq nk$. 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 is it possible to design an algorithm that simultaineously 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 $O((n+m)\log k+\alpha)$ 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 all of the possible combinations of strings in P that imply an occurrence in T. For instance, after we have identified a particularly interesting section in T using Theorem 1. Note that there can be exponentially many of these. Morgante et al. [12] showed how to encode all of these in a graph of size $O(\alpha)$. We can similarly extend our algorithm to produce such an encoding at the cost of using $O(\alpha)$ additional space.

1.3 Technical Overview

The previous work by Morgante et al. [12] and Rahman et al. [18] 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.

[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.

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.

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 [10] for single string matching to multiple strings. The *Aho-Corasick automaton* (AC-automaton) for $\mathcal{P}$, denoted 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 path(s). For each state s there is also a special *failure transition* pointing to the unique state s' such that path(s') is the longest prefix of a pattern in $\mathcal{P}$ matching a proper suffix of 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 occ(s) $\subseteq \mathcal{P}$ of patterns that match a suffix of path(s). Since the patterns in occ(s) share suffixes we can represent occ(s) compactly by storing for s the index of the longest string in occ(s) and a pointer to the state s' such that path(s') is the second longest string if any. In this way we can report 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 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 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

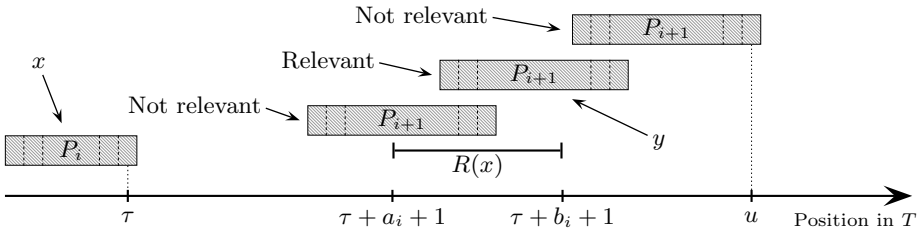

Fig. 1. In this figure x is an occurrence of P_i in T reported at position τ. The first and last occurrence of P_{i+1} start outside $R(x)$ thereby violating the ith gap constraint, so these occurrences are not relevant compared to x. The second occurrence y of P_{i+1} starts in $R(x)$, so if x is itself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

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 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.

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 Fig. 1 for an example. Relevant occurrences are similar to the *valid occurrences* defined in [18]. 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 S be a substring of T matching the VLG pattern $S_1 \cdot g\{a_1, b_1\} \cdot S_2 \cdot g\{a_2, b_2\} \cdots S_k$. Then, startpos($S_{i+1}$) $\in R(S_i)$ 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

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.

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 $\mathit{endpos}(y) \geq \tau$ implies $\mathit{startpos}(y) \geq \tau - |P_i|$. In Fig. 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 range that is not dead. Since the lists are sorted, all remaining ranges in the list are still alive. See Fig. 2 for an example.

3 Analysis

We now show that Algorithm 1 solves the VLG problem in time $O((n+m)\log k + \alpha)$ 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

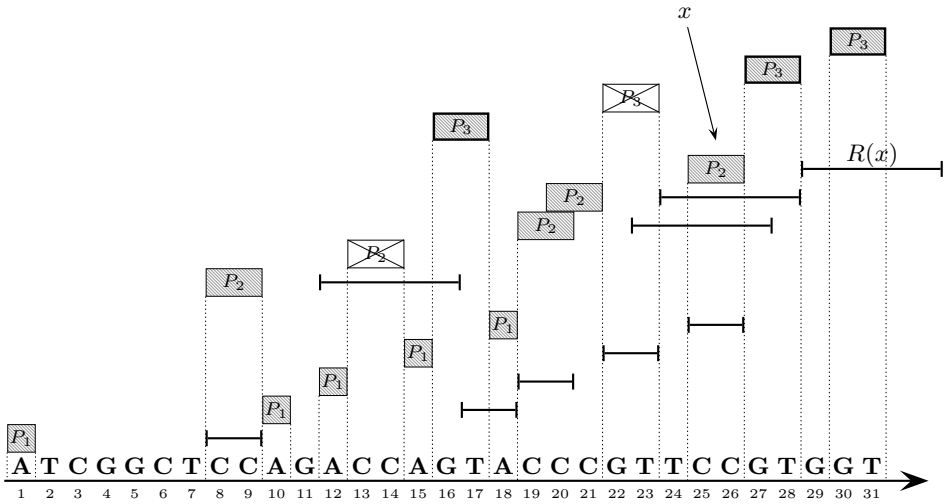

Fig. 2. The occurrences of the subpatterns $P_1 = \text{A}$, $P_2 = \text{CC}$ and $P_3 = \text{GT}$ and the ranges they define in the text T from Example 1. Occurrences which are not relevant are crossed out. The bold occurrences of P_3 are the relevant occurrences of P_k and their end positions 17,28 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 $L_2 = [\,[17;20], [22;23], [25;26]\,]$ and $L_3 = [\,[23,28]\,]$. 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: $L_3 = [\,[23;33]\,]$.

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 *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 $\tau - |P_i| =$ *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 *startpos*$(y) \in R(x)$. By the induction hypothesis there is no range in L_i containing *startpos*(y), since the algorithm only append ranges when a relevant occurrence is found. Consequently, the algorithm correctly determines that y is not relevant.

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 $m = \sum_{i=1}^{k} |P_i|$. 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

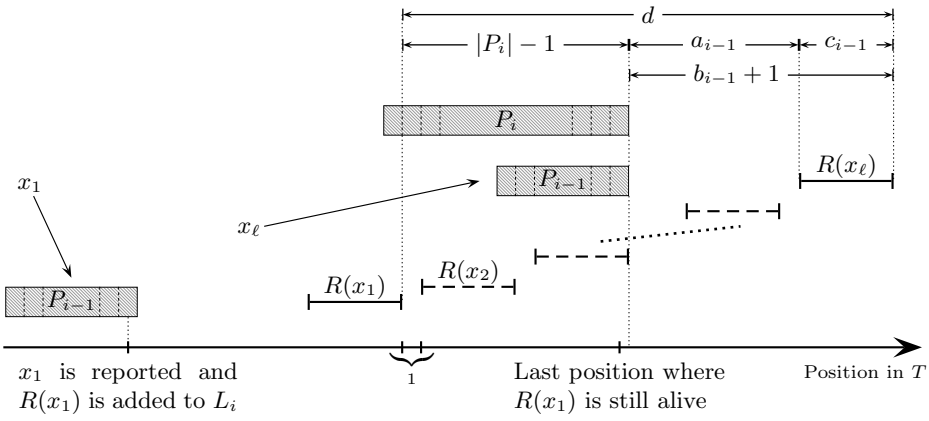

Fig. 3. The worst-case situation where ℓ, the maximum number of ranges are present in L_i. The figure only shows the first and the last occurrence of P_{i-1} (x_1 and x_ℓ) defining the ℓ ranges.

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. Since the AC automata runs in time $O((n+m)\log k + \alpha)$, the total running time is $O((n+m)\log k + \alpha)$.

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.

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 Fig. 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 $O((n+m)\log k + \alpha)$ time and $O(m + A)$ space. This completes the proof of Theorem 1.

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. K. Hofmann, P. Bucher, L. Falquet, and A. Bairoch. The prosite database, its status in. *Nucleic Acids Res*, (27):215–219, 1999.
10. D. E. Knuth, J. James H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.
11. 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.
12. M. Morgante, A. Policriti, N. Vitacolonna, and A. Zuccolo. Structured motifs search. *J. Comput. Bio.*, 12(8):1065–1082, 2005.
13. E. W. Myers. Approximate matching of network expressions with spacers. *J. Comput. Bio.*, 3(1):33–51, 1992.
14. E. W. Myers. A four-russian algorithm for regular expression pattern matching. *J. ACM*, 39(2):430–448, 1992.
15. G. Myers and G. Mehldau. A system for pattern matching applications on biosequences. *CABIOS*, 9(3):299–314, 1993.
16. 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.
17. G. Navarro and M. Raffinot. New techniques for regular expression searching. *Algorithmica*, 41(2):89–116, 2004.
18. 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.
19. K. Thompson. Regular expression search algorithm. *Commun. ACM*, 11:419–422, 1968.