Noname manuscript No.
(will be inserted by the editor)

Approximate Regional Sequence Matching for Genomic Databases

Thanasis Vergoulis · Theodore Dalamagas · Dimitris Sacharidis · Timos Sellis

the date of receipt and acceptance should be inserted later

Abstract Recent advances in computational biology have raised sequence matching requirements that result in new types of sequence database problems. In this work, we introduce an important class of such problems, the *Approximate Regional Sequence Matching (ARSM)* problem. Given a data and a pattern sequence, an ARSM result is an approximate occurrence of a region of the pattern in the data sequence under two conditions. First, the region must contain a predetermined area of the pattern sequence, termed *core*. Second, the allowable deviation between the region of the pattern and its occurrence in the data sequence depends on the length of the region. We propose the `PS-ARSM` method that processes holistically the regions of a pattern, taking advantage of their overlaps to efficiently identify the ARSM results. Its performance is evaluated with respect to existing techniques adapted to the ARSM problem.

1 Introduction

Sequence matching problems (e.g., exact/approximate, local/global alignment) have been extensively studied, and several algorithms, reviewed in [27,9], have been proposed. These problems are very popular as they naturally appear at the heart of many diverse applications. For example, in biological databases, which contain long sequences of symbols (such as nucleotides, amino acids, etc.), sequence matching algorithms help identify homologous (i.e., of similar functionality) biological entities (such as genes, proteins, etc.).

Often, advances in particular research fields introduce complex matching criteria that give rise to *novel sequence matching problems*. Consider the following case from biology. It has been observed that a chemical association, known as binding, of a non-coding RNA sequence (e.g., micro-RNA, short interfering RNA, etc.), termed the *pattern*, with a larger one (e.g., a gene), termed the *data*, usually occurs around a key location of the pattern, called the *core* (e.g., the nucleotides near the start of the micro-RNA [7]). Since laboratory experiments are costly and time-consuming, computational methods to predict bindings based on the previous observation have been proposed (e.g., [18]). In particular, researchers employ a conventional sequence matching algorithm to test whether any super-sequence of the core, termed *region*, matches *approximately* (i.e., a few individual symbols may mismatch) with a subsequence of the data. This process is repeated for each region, and the maximum number of mismatches allowed is set empirically based on the region's length. The larger a region is and the better it matches with the data, the more likely a binding is.

Motivated by this real-life case, this paper generalizes the above matching criteria and introduces the *Approximate Regional Sequence Matching* (ARSM) problem. Assume a *data* sequence S, a *pattern* sequence P, and a *core* (i.e., a subsequence) of P. Briefly, an *ARSM result* is a subsequence of S that approximately matches

T. Vergoulis
NTUA & IMIS, Athena RC
E-mail: vergoulis@dblab.ece.ntua.gr

T. Dalamagas
IMIS, Athena RC
E-mail: dalamag@imis.athena-innovation.gr

D. Sacharidis
IMIS, Athena RC
E-mail: dsachar@imis.athena-innovation.gr

T. Sellis
NTUA & IMIS, Athena RC
E-mail: timos@dblab.ece.ntua.gr

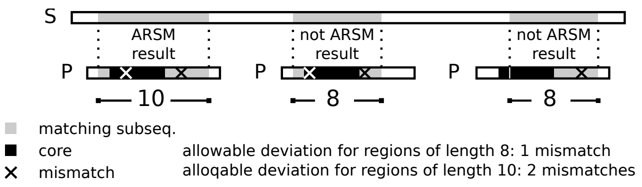

S
ARSM result
not ARSM result
not ARSM result
P
P
P
10
8
8
matching subseq.
core
mismatch
allowable deviation for regions of length 8: 1 mismatch
alloqable deviation for regions of length 10: 2 mismatches

Fig. 1: An example of an ARSM problem.

some subsequence of P under the following conditions: (a) the P subsequence is a *region*, i.e., it encloses the core, and (b) the allowable deviation, in terms of the number of mismatching symbols, between the subsequence of S and the P region grows with the latter's length.

Figure 1 presents an ARSM instance. The top part of the figure depicts the data sequence S, while the bottom shows three copies of the pattern sequence P aligned in different locations under S. The dark shaded part in each P copy corresponds to the core region. On the other hand, the light shaded part depicts a P subsequence (different in each copy) that matches with the corresponding light shaded subsequence of S. In each P subsequence, the number underneath it denotes its length, while a cross indicates a mismatching symbol with respect to S. Furthermore, the allowed number of mismatching symbols is 1 (resp. 2) for regions of length 8 (resp. 10).

Observe that the S subsequence that matches with the second subsequence of P is not an ARSM result because this P subsequence has length 8 and contains more mismatches than allowed. Moreover, neither the S subsequence corresponding to the third subsequence of P is an ARSM result, since the P subsequence does not enclose the core, i.e., it is not a region. On the other hand, the S subsequence corresponding to the first P subsequence *is* an ARSM result as it satisfies both conditions.

The distinctive characteristic of ARSM, compared to other approximate sequence matching problems, is that *multiple* sequences, the regions, are examined for matches under *varying* allowable deviation values. Note that it is possible to extend existing methods to solve the ARSM problem. The naïve approach is to apply a state-of-the-art approximate sequence matching (ASM) algorithm (e.g., [21]) for every possible region. Clearly, this brute-force method is inefficient as it makes no effort to share computation among regions that are overlapping.

A better alternative is to apply a multiple ASM (MASM) algorithm (e.g., [20,8]) that is able to process multiple patterns at a time and exploit their overlaps. Since MASM algorithms are designed to operate on a set of patterns of equal length (see also Section 5), a MASM-based approach must first group regions according to their length, and execute MASM once per group. However, this method cannot take advantage of the overlaps in regions across groups. Besides, for genomic databases, i.e., with small alphabet size (4 symbols), short patterns (a few tens of symbols), and large allowable deviations (around 20% of the pattern length), MASM algorithms are known to suffer [8].

Note that local alignment algorithms (e.g., [30]), which search for matches of all possible pattern subsequences (and thus of the regions as well), cannot be adapted to the ARSM problem for three reasons. First, they require that the allowable deviation is fixed and independent of the subsequence length. Second, the popular state-of-the-art heuristic algorithms (such as BLAST [1]) do not identify all matches. Third, and more importantly, even if exact algorithms are used, some ARSM answers may still be missed (see Section 5 for an explanation).

To overcome the previous limitations, we propose the `PS-ARSM` method, which takes advantage of the prefix and suffix overlaps among the regions. Briefly, our method first determines the data subsequences where the smallest region (the core) matches under the largest possible allowable deviation. Then, based on a set of sound and complete expansion rules, the algorithm progressively expands the data subsequences to derive all ARSM results. Furthermore, we analyze the execution time of `PS-ARSM` and propose cost optimizations. The efficiency of `PS-ARSM`, compared to ASM and MASM based approaches, is validated experimentally on genomic databases.

Outline. Section 2 formally defines the ARSM problem and discusses its characteristics. Section 3 describes the `PS-ARSM` method, and Section 4 presents a detailed experimental evaluation of our approach. Section 5 discusses related work. Finally, Section 6 concludes this paper.

2 The ARSM Problem

Section 2.1 formally introduces the ARSM problem, and Section 2.2 studies its key characteristics that form the basis of our ARSM method (described in Section 3).

2.1 Problem Definition

Consider an alphabet Σ. In the remainder of this paper, each sequence $S \in \Sigma^*$. $|S|$ denotes the length of sequence S. $S_{[i]}$ corresponds to the i-th symbol in S,

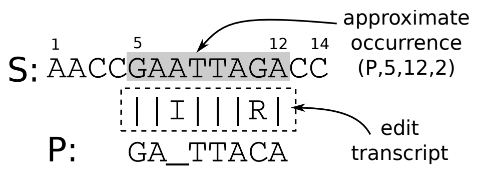


Fig. 2: An approximate occurrence of GATTACA in AACCGAATTACACC.

while $S_{[i,j]}$ to the subsequence of S that starts at the i-th and ends at the j-th symbol. We use the notation $S_{[i,j]} \sqsubset S$ to indicate that $S_{[i,j]}$ is a subsequence of S.

Given two sequences, we call *transcript* an ordered set τ of *edit operations* to transform one sequence to the other. Typically, the following edit operations are allowed for a sequence S:

– Insert (`I`) a symbol into S,
– Delete (`D`) a symbol of S,
– Replace (`R`) a symbol of S with another,
– Match (`M`), i.e., preserve, a symbol of S.

The *cost* $c(\tau)$ of a transcript is the number of `I`, `D`, and `R` operations it contains.

Consider two sequences S and P, called data sequence and pattern sequence, respectively. One can always find a transcript that transforms P into any subsequence $S_{[i,j]}$ of the data sequence. We say that the pattern P *occurs* in the data S at location $[i, j]$ with transcript cost ϵ, if there is a transcript τ that transforms P into $S_{[i,j]}$ having cost $c(\tau) = \epsilon$. We use the notation (P, i, j, ϵ) to indicate this *occurrence*.

Figure 2 shows an occurrence $(P, 5, 12, 2)$ of pattern $P =$ `GATTACA` in $S =$ `AACCGAATTAGACC` at location $[5, 12]$ with transcript cost $\epsilon = 2$. Indeed, according to the transcript $\tau =$ `MMIMMMRM`, P can be transformed into $S_{[5,12]}$ by inserting (`I`) symbol `A` between $P_{[2]}$ and $P_{[3]}$, and replacing (`R`) $P_{[6]} =$ `C` with `G`; all other symbols remain unchanged (`M`).

Note that there can be more than one transcripts to transform P in $S_{[i,j]}$ with the same cost ϵ. For instance, in the example of Figure 2, the transcript $\tau' =$ `MIMMMMRM` has the same cost with the depicted one.

A pattern can occur at a particular location in the data with various transcript costs. An occurrence (P, i, j, ϵ) is called *minimal* if it has the lowest possible cost, i.e., there exists no other occurrence (P, i, j, ϵ') such that $\epsilon' < \epsilon$. For example, the occurrence $(P, 5, 12, 2)$ in Figure 2 is minimal.

Given a pattern sequence P and a location $[a, b]$ in P termed *core* ($1 \leq a \leq b \leq |P|$), a subsequence $R = P_{[i,j]}$ of P is called *region* if $i \leq a$ and $j \geq b$. Two special regions are the *core region* $P_{[a,b]}$ and the *pattern region* $P_{[1,|P|]}$. We next introduce the ARSM problem.

Definition 1 (ARSM Problem) Given a data sequence S, a pattern P, its core $[a, b]$, and a monotonically increasing threshold function $\mathcal{K} : \mathbb{N} \to \mathbb{N}$, the *ARSM problem* is to retrieve the minimal occurrences (R, i, j, ϵ) of each region R of P, such that: (1) $\epsilon \leq \mathcal{K}(|R|)$, and (2) no other minimal occurrence (R', i, j, ϵ'), where $R \sqsubset R'$, has $\epsilon' \leq \mathcal{K}(|R'|)$. □

The first constraint specifies that the allowed cost for a region occurrence depends on the size of the region and is given by the threshold function. Larger regions are allowed to have larger cost. The second constraint implies that if two different regions R, R' occur at the same location $[i, j]$ of the data sequence S, then only the occurrence of the largest region is returned. We call all those retrieved minimal occurrences *ARSM results.*

Figure 3 illustrates an instance of the ARSM problem in which the data sequence S has 19 symbols and the pattern P has 10. The core is the location $[5, 8]$ of the pattern. In this instance, there exist 15 possible regions labelled $R1$ through $R15$. Figure 3 also depicts the values of the threshold function $\mathcal{K}$ for all possible region lengths (4 up to 10).

2.2 ARSM Characteristics

Section 2.2.1 presents some key observations regarding occurrences of overlapping regions. Recall that regions are highly overlapping (see Figure 3), since each one is a subsequence of the pattern and a supersequence of the core. Then, Section 2.2.2 exploits these observations to introduce a set of expansion rules that construct the set of minimal occurrences of a region from those of a smaller one.

2.2.1 Overlapping Occurrences

Consider a data sequence S and a pattern P. Assuming that P occurs at location $[i, j]$ of S, the next two lemmas show how this occurrence is related to an occurrence of P at locations $[i, j + 1]$ and $[i - 1, j]$. Intuitively, an edit operation `I` can be appended to the transcript to accommodate for the extra symbol of the data sequence at location $j + 1$ or $i - 1$.

Lemma 1 *If (P, i, j, ϵ) is an occurrence of P in data sequence S and τ is one of its transcripts, then $(P, i, j + 1, \epsilon + 1)$ is also an occurrence of P in S and τ`I` is one of its transcripts.*

Proof The transcript τ`I` contains the same edit operations as τ, and an additional `I` operation. Therefore, $c(\tau\mathtt{I}) = c(\tau) + 1 = \epsilon + 1$. As τ transforms P into $S_{[i,j]}$,

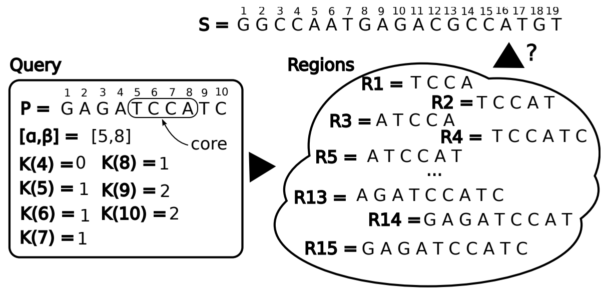


Fig. 3: ARSM example

the last I in τI inserts $S_{[j+1]}$ at the end of P. Therefore, τI, having transcript cost $\epsilon+1$, transforms P into $S_{[i,j+1]}$, i.e., $(P, i, j+1, \epsilon+1)$ is an occurrence of P in S and τI is one of its transcripts.

Lemma 2 *If (P, i, j, ϵ) is an occurrence of P in data sequence S and τ is one of its transcripts, then $(P, i-1, j, \epsilon+1)$ is also an occurrence of P in S and Iτ is one of its transcripts.*

Proof Similar to that of Lemma 1.

Next, consider two pattern sequences, P and $P\gamma$, where the latter is obtained by appending symbol $\gamma \in \Sigma$ at the end of the former. This represents the case where regions share the same prefix and differ by one symbol, e.g., regions $R2$ and $R4$ in Figure 3. Assuming P occurs at location $[i, j]$ in S, the next lemma shows how this occurrence is related to occurrences of $P\gamma$ at locations $[i, j]$ and $[i, j+1]$. Intuitively, an edit operation (D, R, or M) can be appended to the transcript to accommodate for the extra symbol γ of the pattern sequence.

Lemma 3 *If (P, i, j, ϵ) is an occurrence of P in data sequence S and τ is one of its transcripts, then:*

1. *$(P\gamma, i, j, \epsilon+1)$ is an occurrence of $P\gamma$ in S and τD is one of its transcripts,*
2. *$(P\gamma, i, j+1, \epsilon)$ is an occurrence of $P\gamma$ in S and τM is one of its transcripts, if $S_{[j+1]} = \gamma$,*
3. *$(P\gamma, i, j+1, \epsilon+1)$ is an occurrence of $P\gamma$ in S and τR is one of its transcripts, if $S_{[j+1]} \neq \gamma$.*

Proof We prove case 1; cases 2 and 3 can be proven similarly. The transcript τD contains the same edit operations as τ, and an additional D operation. Therefore, $c(\tau\mathtt{D}) = c(\tau) + 1 = \epsilon + 1$. As τ transforms P into $S_{[i,j]}$ and the last D in τD just deletes γ from $P\gamma$, then τD transforms $P\gamma$ into $S_{[i,j]}$. Therefore, $(P\gamma, i, j, \epsilon+1)$ is an occurrence of $P\gamma$ in S and τD is one of its transcripts.

Finally, consider two pattern sequences, P and θP, where the latter is obtained by appending symbol $\theta \in \Sigma$ at the beginning of the former. This represents the case where regions share the same suffix and differ by one symbol, e.g., regions $R1$ and $R3$ in Figure 3. Assuming P occurs at location $[i, j]$ in S, the next lemma shows how this occurrence is related to occurrences of θP at locations $[i, j]$ and $[i-1, j]$. As before, an edit operation (D, R, or M) can be appended to the transcript to accommodate for the extra symbol θ of the pattern sequence.

Lemma 4 *If (P, i, j, ϵ) is an occurrence of P in data sequence S and τ is one of its transcripts, then:*

1. *$(\theta P, i, j, \epsilon+1)$ is an occurrence of θP in S and Dτ is one of its transcripts,*
2. *$(\theta P, i-1, j, \epsilon)$ is an occurrence of θP in S and Mτ is one of its transcripts, if $S_{[i-1]} = \theta$,*
3. *$(\theta P, i-1, j, \epsilon+1)$ is an occurrence of θP in S and Rτ is one of its transcripts, if $S_{[i-1]} \neq \theta$.*

Proof Similar to that of Lemma 3.

2.2.2 Prefix and Suffix Expansions

Based on the lemmas of the previous section, we next show that it is possible to construct the set of minimal occurrences of a region from those of a smaller one. Assume a data sequence S, a pattern P, and a cost threshold k. Let $\mathcal{O}$ be the set of minimal occurrences of P in S with transcript cost not more than k. In the following, we describe a set of expansion rules that, when applied to $\mathcal{O}$, produce the minimal occurrences of patterns $P\gamma$ and θP, where $\gamma, \theta \in \Sigma$. We describe two sets of expansion rules: prefix and suffix expansion rules.

First, we present the prefix expansion rules. Consider the case of pattern $P\gamma$, which has P as prefix.

Definition 2 (Prefix Expansion) The *prefix expansion* of $\mathcal{O}$ with symbol $\gamma \in \Sigma$, denoted as $\mathcal{O}^\gamma$, is a set of occurrences of pattern $P\gamma$ in S, with cost not more than k, derived according to the following expansion rules.

For each $(P, i, j, \epsilon) \in \mathcal{O}$:

1. If $\epsilon+1 \leq k$, insert into $\mathcal{O}^\gamma$ the occurrences $(P\gamma, i, j+x, \epsilon+x+1)$ for all $0 \leq x \leq k-\epsilon-1$.
2. If $\epsilon \leq k$ and $S_{[j+1]} = \gamma$, insert into $\mathcal{O}^\gamma$ the occurrences $(P\gamma, i, j+x+1, \epsilon+x)$ for all $0 \leq x \leq k-\epsilon$.
3. If $\epsilon+1 \leq k$ and $S_{[j+1]} \neq \gamma$, insert into $\mathcal{O}^\gamma$ the occurrences $(P\gamma, i, j+x+1, \epsilon+x+1)$ for all $0 \leq x \leq k-\epsilon-1$.

During occurrence insertion, if another in $\mathcal{O}^\gamma$ occurs at the same location, keep the one with the smallest transcript cost. □

Intuitively, these rules apply cases 1, 2, or 3 of Lemma 3, respectively, to occurrence (P, i, j, ϵ), and, then, apply Lemma 1 repeatedly (once per x value so as not to exceed the error threshold k) to each derived occurrence of $P\gamma$.

As an example, consider the data sequence S of Figure 3 and let $P =$ `GCCA`, $\gamma =$ `T`. $(P, 13, 16, 0)$ is one minimal occurrence of P in S with cost not more than $k = 1$. Expansion rule 1 on occurrence $(P, 13, 16, 0)$ produces $(P\gamma, 13, 16, 1)$, while rule 2 produces $(P\gamma, 13, 17, 0)$ and $(P\gamma, 13, 18, 1)$. Note that expansion rule 3 does not apply, since $S_{[17]} = \gamma =$ `T`.

The next theorem shows that the prefix expansion rules are *sound*, i.e., they produce occurrences of $P\gamma$ that are minimal, and *complete*, i.e., they produce all minimal occurrences of $P\gamma$.

Theorem 1 *If $\mathcal{O}$ is the set of all minimal occurrences of P in S with transcript cost not more than k, then its prefix expansion $\mathcal{O}^{\gamma}$ is the set of all minimal occurrences of $P\gamma$ in S with transcript cost not more than k.*

Proof Let $\mathcal{O}'$ be the set of all minimal occurrences of $P\gamma$ in S with transcript cost not more than k. We first show that $\mathcal{O}' \subseteq \mathcal{O}^{\gamma}$ by contradiction.

Suppose there is a minimal occurrence $(P\gamma, i, j, \epsilon)$ of $P\gamma$ with $\epsilon \leq k$ which does not appear in $\mathcal{O}^{\gamma}$. Let τ be any of the edit transcripts of this occurrence. There are four cases based on the last operation in τ. Let τ' denote the transcript obtained by omitting this last operation.

Assume the last operation is `D`, i.e., $\tau = \tau'$`D`. It is easy to see that $c(\tau') = \epsilon - 1$ and that $(P, i, j, \epsilon - 1)$ is an occurrence of P. We show that this occurrence is minimal.

Suppose otherwise. Then, there exists a transcript $\tau*$ that corresponds to an occurrence $(P, i, j, c(\tau^*))$, where $c(\tau^*) < c(\tau') = \epsilon - 1$. According to case 1 of Lemma 3, $(P\gamma, i, j, c(\tau^*) + 1)$ is an occurrence of $P\gamma$ with transcript τ^*`D` and cost $c(\tau^*)+1 < \epsilon$. However, this is not possible, as the minimal occurrence of $P\gamma$ at location $[i, j]$ has cost ϵ, as assumed. Hence, $(P, i, j, \epsilon - 1)$ is a minimal occurrence of P.

Consequently, according to the first rule (for $x = 0$), $(P, i, j, \epsilon - 1)$ is expanded to occurrence $(P\gamma, i, j, \epsilon)$. However, the latter was assumed to not appear in $\mathcal{O}^{\gamma}$, which is a contradiction.

Using similar reasoning, one can show that this contradiction appears for the cases when the last operation is `R` and `M`, using cases 3 and 2 of Lemma 3, respectively.

We have to show a contradiction for the last case, when the last operation in transcript τ is `I`. Let y be the largest integer such that the last y operations in τ are all insertions. Further, let τ_1 be the transcript obtained from τ by omitting those last y operations. Observe that $(P\gamma, i, j - y, c(\tau_1))$ has to be a minimal occurrence of $P\gamma$ at location $[i, j - y]$ with transcript cost $c(\tau_1) = \epsilon - y$; otherwise, one can construct an occurrence of $P\gamma$ at location $[i, j]$ with cost lower than the minimum ϵ.

Depending on the last operation in τ_1 (which cannot be `I`), one can construct a minimal occurrence of P from $(P\gamma, i, j - y, c(\tau_1))$ in a manner similar to the three cases examined before. Then, applying the corresponding expansion rule setting $x = y$, we obtain that $(P\gamma, i, j, \epsilon)$ appears in $\mathcal{O}^{\gamma}$, i.e., a contradiction. Therefore, $\mathcal{O}' \subseteq \mathcal{O}^{\gamma}$.

Finally, we show that $\mathcal{O}' \supseteq \mathcal{O}^{\gamma}$. Suppose otherwise, i.e., there exists an occurrence $(P\gamma, i, j, \epsilon)$ of $\mathcal{O}^{\gamma}$ that it is not in $\mathcal{O}'$. Since this occurrence is not minimal, there must exist another, say $(P\gamma, i, j, \epsilon') \in \mathcal{O}'$, with $\epsilon' < \epsilon$. We have already shown that $\mathcal{O}' \subseteq \mathcal{O}^{\gamma}$, which implies that $(P\gamma, i, j, \epsilon')$ is also in $\mathcal{O}^{\gamma}$. As a result, both $(P\gamma, i, j, \epsilon)$ and $(P\gamma, i, j, \epsilon')$ are in $\mathcal{O}^{\gamma}$. This is a contradiction because the expansion rules dictate that only the occurrence with the smallest cost among those occurring at the same location is allowed in $\mathcal{O}^{\gamma}$.

Finally, consider the case of pattern θP, which has P as suffix.

Definition 3 (Suffix Expansion) The *suffix expansion* of $\mathcal{O}$ with symbol $\theta \in \Sigma$, denoted as ${}^{\theta}\mathcal{O}$, contains a set of occurrences of pattern θP in S with cost not more than k, and is derived according to the following expansion rules.

For each $(P, i, j, \epsilon) \in \mathcal{O}$:

1. If $\epsilon + 1 \leq k$, insert into ${}^{\theta}\mathcal{O}$ the occurrences $(\theta P, i - x, j, \epsilon + x + 1)$ for all $0 \leq x \leq k - \epsilon - 1$.
2. If $\epsilon \leq k$ and $S_{[i-1]} = \theta$, insert into ${}^{\theta}\mathcal{O}$ the occurrences $(\theta P, i - x - 1, j, \epsilon + x)$ for all $0 \leq x \leq k - \epsilon$.
3. If $\epsilon + 1 \leq k$ and $S_{[i-1]} \neq \theta$, insert into ${}^{\theta}\mathcal{O}$ the occurrences $(\theta P, i - x - 1, j, \epsilon + x + 1)$ for all $0 \leq x \leq k - \epsilon - 1$.

During occurrence insertion, if another in ${}^{\theta}\mathcal{O}$ occurs at the same location, keep the one with the smallest transcript cost. □

Intuitively, these rules apply cases 1, 2, or 3 of Lemma 4, respectively, to occurrence (P, i, j, ϵ), and, then, apply Lemma 2 repeatedly (once per x value so as not to exceed the error threshold k) to each derived occurrence of θP. The next theorem shows the soundness and completeness of the suffix expansion rules.

Theorem 2 *If $\mathcal{O}$ is the set of all minimal occurrences of P in S with transcript cost not more than k, then its suffix expansion ${}^{\theta}\mathcal{O}$ is the set of all minimal occurrences of θP in S with transcript cost not more than k.*

Proof Similar to that of Theorem 1.

Note that consecutive applications of the prefix and suffix expansion rules can produce the minimal occurrences of a pattern from the minimal occurrences of one of its subsequences, as shown in the next theorem.

Theorem 3 *Given two patterns P, P' such that $P \sqsubset P'$, and the set $\mathcal{O}$ of all minimal occurrences of P in S with transcript cost not more than k, it is possible to construct the set of all minimal occurrences of P' in S with transcript cost not more than k', for any $k' \leq k$.*

Proof Observe that for any pattern the set of its occurrences with cost not more than k is a superset of the set of its occurrences with cost not more than k', where $k' \leq k$. Therefore, we only need to prove that the set of minimal occurrences of P' with transcript cost not more than k can be obtained from $\mathcal{O}$.

Since $P \sqsubset P'$, there exists a sequence of patterns $P_1, \ldots, P_n$, such that $P_1 = P$, $P_n = P'$, and either $P_{i+1} = P_i\gamma$ or $P_{i+1} = \theta P_i$ holds, where $\gamma, \theta \in \Sigma$. Due to Theorems 1 and 2, an application of the appropriate (prefix if $P_{i+1} = P_i\gamma$, suffix otherwise) expansion rules to the set of minimal occurrences of P_i constructs the set of all minimal occurrences of P_{i+1}. After successive applications, the required set can be constructed.

3 The PS-ARSM Method

The *Prefix-Suffix ARSM* (`PS-ARSM`) method exploits the overlaps among regions and applies the expansion rules of Section 2.2.2 to efficiently produce all ARSM results. The key idea is to initially determine the minimal occurrences for the smallest possible region, the core, and then progressively expand them to construct the minimal occurrences for all regions. Special care is required so that the produced occurrences obey the two requirements set in Definition 1. Note that since all occurrences produced by our method are minimal, we drop the characterization minimal in the remainder of this paper.

We first introduce some important concepts in Section 3.1. Then, we describe the `PS-ARSM` algorithm in Section 3.2, and detail its implementation in Section 3.3. Finally, we present a cost analysis of `PS-ARSM` and propose an optimization in Section 3.4. For ease of reference, we include the most common symbols and their definitions in Table 1.

Table 1: Common notation.

Symbol	**Definition**
Σ	Alphabet
S	Data sequence
P	Pattern
R	A region of P
C	Core of P
a	Start position of C in P
b	End position of C in P
$\mathcal{K}()$	Threshold function
c	A suffix chain in the region lattice
sc_num	Number of suffix chains
sc_length	Length of the suffix chains

3.1 Region Lattice

`PS-ARSM` operates on the *region lattice* induced by the subsequence relation $\sqsubset$. Figure 4 presents the region lattice for the example of Figure 3. The top left region $R1$ corresponds to the core, while the bottom right $R15$ to the pattern. A horizontal (resp. vertical) arrow from a region to another one implies that the former is a suffix (resp. prefix) of the latter.

A *head* is a region such that none of its suffixes, except itself, are regions. A *tail* is a region such that it is not the suffix of any other region. In Figure 4, there exist three heads, $R1$, $R2$ and $R4$, and three tails, $R12$, $R14$ and $R15$. The heads and tails of a lattice are totally ordered based on the $\sqsubset$ relation. The first head is the core, while the last tail is the pattern.

A *suffix chain* is a totally ordered set of regions that contains a tail and all its suffixes. The suffix chains of a lattice are ordered according to the rank of their tail. There exist three suffix chains in Figure 4, labelled $c1$, $c2$, $c3$, each corresponding to a row of the lattice. Observe that the smallest region in a suffix chain is a head and the largest is a tail; e.g., in chain $c1 = \{R1, R3, R6, R9, R12\}$, $R1$ and $R12$ are its head and tail, respectively.

3.2 Algorithm Description

`PS-ARSM` consists of three execution phases. We next describe these phases in detail.

Phase 1. In this phase, `PS-ARSM` determines the occurrences of the core with transcript cost not more that $\mathcal{K}(|P|)$. Note that the cost threshold $\mathcal{K}(|P|)$ is selected

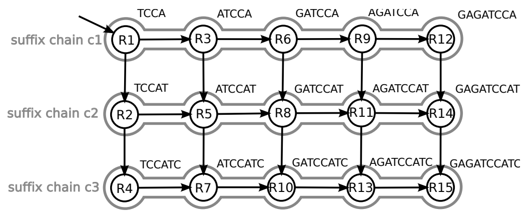


Fig. 4: The region lattice for the ARSM example.

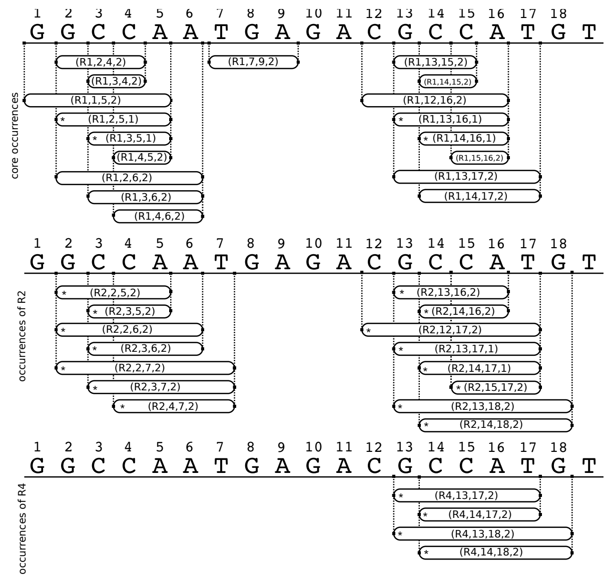


Fig. 5: Core occurrences, head occurrences and seeds for suffix chains c_1, c_2, and c_3.

so that no ARSM result, produced by expansions of the core occurrences, is missed, as explained in the following.

Recall that an occurrence of a region R can only be an ARSM result if its cost is not more than $\mathcal{K}(|R|)$ (first requirement in Definition 1). Since the pattern P is the largest region and $\mathcal{K}$ is monotonically increasing, $\mathcal{K}(|P|)$ is the highest cost any ARSM result is allowed to have. From Theorem 3, it follows that any ARSM result must be among the expansions of the core occurrences with the loosest possible cost threshold, i.e., $\mathcal{K}(|P|)$.

For the ARSM example shown in Figure 3 and the corresponding lattice in Figure 4, the first phase of `PS-ARSM` computes the core occurrences, i.e., those of region $R1 = $ `TCCA`, with cost at most $\mathcal{K}(|P|) = \mathcal{K}(10) = 2$. All these occurrences are depicted at the top of Figure 5 as oval boxes aligned with respect to the data sequence S. For instance, $(R1, 2, 4, 2)$ corresponds to an occurrence of the core $R1$ at location $[2, 4]$ in S.

Phase 2. In this phase, `PS-ARSM` first applies the prefix expansion rules on the core occurrences to produce the occurrences of all heads. E.g., in the lattice of Figure 4, `PS-ARSM` prefix-expands the core occurrences to construct the occurrences of region $R2 = $ `TCCAT`, shown at the middle of Figure 5. The resulting head occurrences are then expanded to obtain those of $R4 = $ `TCCATC`, shown at the bottom of Figure 5.

Next, `PS-ARSM` filters the head occurrences of each suffix chain to provide the appropriate input to phase 3. Consider a chain c, and let R^c_1 and R^c_n be its head and tail, respectively. Briefly, the goal of phase 3 is to produce the occurrences of any region in c by expanding its head occurrences. Note that all produced occurrences have cost at most $\mathcal{K}(|P|)$, as they are expansions of the core occurrences. However, observe that $\mathcal{K}(|R^c_n|)$, which is not more than $\mathcal{K}(|P|)$, is the highest cost any occurrence of a region in c is allowed to have (see Definition 1). Therefore, from Theorem 3, it follows that only head occurrences with cost not more than $\mathcal{K}(|R^c_n|)$ should be suffix-expanded in phase 3. We refer to these occurrences of R^c_1 as the *seeds* of the suffix chain c. For example, the head occurrences in Figure 5 that are also seeds are marked with an asterisk; e.g., $(R2, 2, 5, 2)$ is a seed of suffix chain $c2$. Thus, phase 2 filters out non-seeds from head occurrences to provide the phase 3 input.

Phase 3. In this phase, `PS-ARSM` produces the ARSM results. The algorithm operates holistically on all chains, but for clarity we only describe the procedure for a single chain c. For each region along the chain, starting from the head R^c_1 and ending at the tail R^c_n, `PS-ARSM` performs the following tasks.

Suppose R^c_i is the current region. `PS-ARSM` first suffix-expands the occurrences of the previous region R^c_{i-1} in the chain to produce the occurrences of R^c_i. These expanded occurrences are called *candidates*, and the candidates of R^c_1 are the seeds. Then, `PS-ARSM` enforces the two requirements of Definition 1. For the first, it excludes candidates with cost more than $\mathcal{K}(|R^c_i|)$; the remaining are to be inserted in the result set. However, during insertion, `PS-ARSM` removes any occurrence (either a candidate, or one already in the result set) that violates the second requirement. It is important to note that all candidates for R^c_i (i.e., not only those inserted in the result set) are required to obtain the candidates for the next region R^c_{i+1} in the chain.

Figure 6 illustrates the third phase for all the suffix chains of the lattice shown in Figure 4. Consider suffix chain c_2. Its head is $R^{c2}_1 = R2$ and the tail is $R^{c2}_n = R14$. The seeds of this chain, i.e., the occurrences of $R2$ with cost at most $\mathcal{K}(|R14|) = \mathcal{K}(9) = 2$, have been determined in the second phase. Assume that the currently examined region is $R5$. Its four candidates (see the second part of Figure 6) are produced by the suffix expansion of $R2$ candidates (at the middle of the first part of Figure 6). Then `PS-ARSM` considers only those candidates with cost 1 (since $\mathcal{K}(|R5|) = \mathcal{K}(6) = 1$) for insertion in the result set. In our example, this is an empty set.

Pseudocode. Figure 7 presents the `PS-ARSM` pseudocode. The algorithm takes as input the data sequence S, the pattern P, the core $[a, b]$ and the threshold function $\mathcal{K}$, and outputs the ARSM results. In the first phase, `PS-ARSM` applies an ASM algorithm to compute

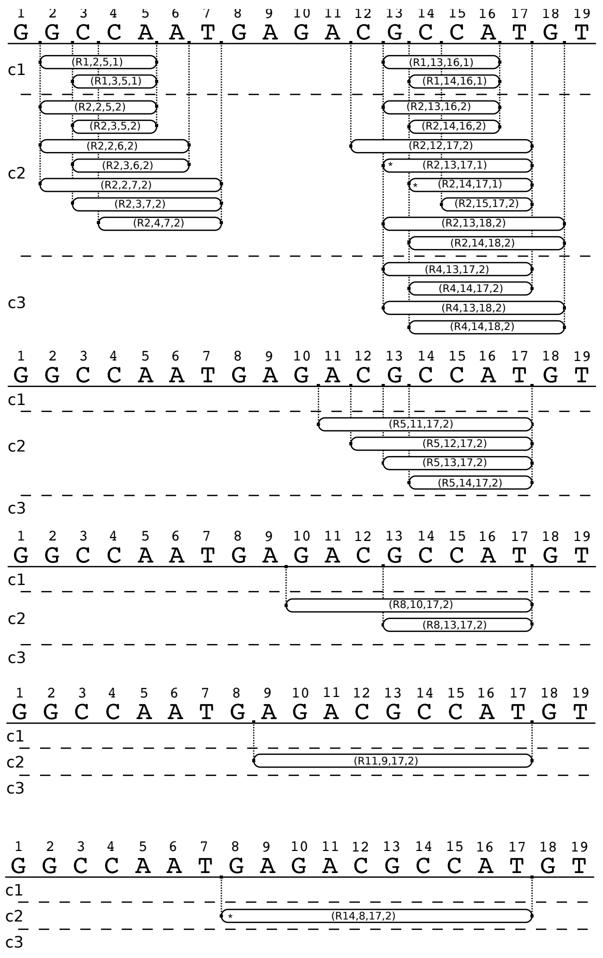


Fig. 6: Seeds, candidates and temporal results.

the core occurrences (line 1); details are discussed in Section 3.3.2.

In the second phase (lines 2–6), PS-ARSM constructs the head occurrences. The occurrences of the first head are those of the core (line 2). The occurrences of each other head are obtained from those of the previous head (line 4). The pExp method implements the prefix expansion rules. Then, PS-ARSM identifies the seeds for the corresponding suffix chain, by invoking the getSeeds method (line 5). Both methods are discussed in Section 3.3.3.

In the third phase (lines 7–14), PS-ARSM computes the ARSM results. For clarity, pseudocode presents the required steps independently for each suffix chain c_j. However, these steps are performed holistically for the seeds of all suffix chains (see Section 3.3.4 for details). Back to pseudocode, PS-ARSM first constructs the candidates, i.e., the occurrences of each region in the chain. Note that the candidates of the head are the seeds of the chain (line 8). The candidates for each other region are obtained from those of the previous region

```
PS-ARSM()
  Input: $S$, $P$, $[a,b]$, $\mathcal{K}$
  Output: $results$
begin
  # Phase 1
01.  $core\_occs \leftarrow$ ASMfull($S$, $P_{[a,b]}$, $\mathcal{K}(|P|)$)
  # Phase 2
02.  $head\_occs_{c_1} \leftarrow core\_occs$
03.  foreach suffix chain $c_j$
04.      if $(j>1)$ then $head\_occs_{c_j} \leftarrow$ pExp($head\_occs_{c_{j-1}}$, $R_1^{c_j}$)
05.      $seeds_{c_j} \leftarrow$ getSeeds($head\_occs_{c_j}$, $\mathcal{K}(|c_j.tail|)$)
06.  end
  # Phase 3
07.  foreach suffix chain $c_j$
08.  $cands_1 \leftarrow seeds_{c_j}$
09.      foreach region $R_i^{c_j}$ of $c_j$
10.          if $(i>1)$ then $cands_i \leftarrow$ sExp($cands_{i-1}$, $R_i^{c_j}$)
11.          $tmp \leftarrow$ sieve($cands_i$, $\mathcal{K}(|R_i^{c_j}|)$)
12.          insert($results$, $tmp$)
13.      end
14.  end
end.
```

Fig. 7: The PS-ARSM algorithm.

(line 10). In particular, the sExp method (presented in Section 3.3.4) implements the suffix expansion rules. Then, among the candidates of region $R_i^{c_j}$, those with cost not more than $\mathcal{K}(|R_i^{c_j}|)$ are identified by the sieve method (line 11 and Section 3.3.4). Finally, the remaining occurrences are inserted in the result set, enforcing the second requirement of Definition 1 (line 12).

Correctness. PS-ARSM applies Theorem 3, i.e., it implements prefix and suffix rules, which are sound and complete as we proved. The previous guarantees that PS-ARSM will return all the ARSM results.

3.3 Implementation

Section 3.3.1 presents the main data structures used by PS-ARSM. Then, Sections 3.3.2, 3.3.3 and 3.3.4 detail Phases 1, 2, and 3, respectively.

3.3.1 Maintaining head occurrences and candidates

An important observation of PS-ARSM is that prefix (resp. suffix) expanding a set of occurrences that end (resp. start) at the same position in the data sequence, requires the same computations. To understand this, refer to Figure 5, and consider the core occurrences $(R1, 1, 5, 2)$ and $(R1, 2, 5, 1)$, which both end at position 5. We describe the operations necessary to prefix expand these occurrences and obtain the head occurrences of region $R2$. First, observe that $R2 = R1\,\texttt{T}$, and that the next symbol in the data sequence, i.e., at position 6, is A. Then, applying Lemma 3 for $P = R1$, $i \in \{1, 2\}$ $j = 5$, $\epsilon \in \{2, 1\}$, and since $S[j+1] = \texttt{A} \neq \texttt{T} = \gamma$, we obtain that each core occurrence results in two head occurrences: $(R2, 1, 5, 3)$, $(R2, 1, 6, 3)$ from the first, and $(R2, 2, 5, 2)$, $(R2, 2, 6, 2)$ from the second. Observe that these can be succinctly represented as the two head occurrences $(R2, i, 5+\underline{0}, \epsilon+\underline{1})$ and $(R2, i, 5+\underline{1}, \epsilon+\underline{1})$, for $i \in \{1, 2\}$, $\epsilon \in \{2, 1\}$. It is important to notice here that the information we have underlined in the previous representation (i.e., +0, +1 for the end position, and +1, +1 for the cost) completely describes how the head occurrences are constructed from the core occurrences.

Based on this observation, PS-ARSM employs data structures that serve two goals: (1) they compactly represent occurrences, and (2) they facilitate prefix and suffix expansions by avoiding redundant computations.

Hash table head_occs. The hash table head_occs is used to compute and store the head occurrences during the second phase of PS-ARSM. Initially, it contains the core occurrences, i.e., the head occurrences of the first suffix chain. Then, its entries are updated $sc\_num - 1$ times, where sc_num is the number of suffix chains in the region lattice. After the $(i-1)$-th update, head_occs contains the head occurrences for the i-th suffix chain.

In what follows, we describe the head_occs contents for suffix chain c_i; the update procedure is explained in Section 3.3.3. head_occs consists of key-value entries. A *key* corresponds to the end position of a core occurrence in the data sequence. The *value* for key j, denoted as head_occs[j], is a composite value with information about the head occurrences of c_i produced by the prefix-expansion of core occurrences ending at position j. In particular, this composite value consists of:

- core_occs: the list of core occurrences ending at position j, ordered by their transcript cost.
- add_cost: an array where the x-th entry denotes the additional transcript cost required for the expansion of any core occurrence in core_occs to a head occurrence of c_i ending at position $j + x$.

Note that the length of the add_cost array is $sc\_num + \mathcal{K}(|P|)$, as this value is the farthest a core occurrence can be expanded to the right while its transcript cost remains not more than $\mathcal{K}(|P|)$.

Observe that head_occs indirectly describes the head occurrences of suffix chain c_i. That is, it stores the core occurrences (core_occs), and how to expand them (add_cost) so as to produce the head occurrences. We illustrate the above using an example. Assume the lattice of Figure 4 and consider head_occs after the first update, i.e., containing the head occurrences of suffix chain c_2. Figure 8 depicts the contents of entry head_occs[5]. core_occs contains all the core occurrences ending at position 5 in the data sequence S. Note that an add_cost array is represented as two rows: the

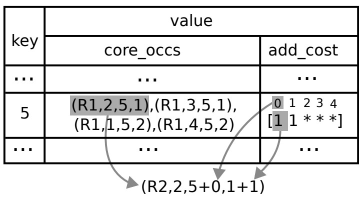


Fig. 8: An excerpt of the head_occs hash table, containing the head occurrences of suffix chain c_2.

lower shows the contents, while the upper presents the indices of this zero-based array. The depicted add_cost informs us that head occurrences of c_2 (produced by any core occurrence in core_occs) ending at positions 5+0 = 5 and 5+1 = 6, have additional cost of 1. Moreover, any head occurrence ending at subsequent positions, e.g., $5 + 2 = 7$, has an additional cost larger than $\mathcal{K}(|P|)$, denoted as $*$ in Figure 8. Therefore, the head occurrences in head_occs[5] are $(R2, 2, 5, 2)$, $(R2, 2, 6, 2)$, $(R2, 3, 5, 2)$ and $(R2, 3, 6, 2)$, which are all head occurrences of c_2 that are expansions of core occurrences ending at position 5.

Hash table cands. The hash table cands is used to compute and store the candidates during the third phase of PS-ARSM. Initially, it contains the seeds of all the suffix chains, i.e., the candidates for all the heads. Then, its entries are updated $sc\_length-1$ times, where sc_length is the length of any suffix chain in the region lattice. After the $(i + 1)$-th update, cands contains the candidates for the i-th regions of all suffix chains.

In what follows, we describe the contents of cands for the i-th regions of all suffix chains; the update procedure is explained in Section 3.3.4. cands consists of key-value pairs. A *key* corresponds to the start position of a seed in the data sequence. The *value* for key j, denoted as cands[j], is a composite value with information about the candidates produced by the suffix-expansion of seeds starting at position j. In particular, this composite value consists of:

- seeds: the list of seeds starting at position j, ordered by their transcript cost.
- add_cost: an array where the x-th entry denotes the additional transcript cost required for the expansion of any seed in seeds to a candidate starting at position $j - x$.

Note that the length of the add_cost array is $sc\_length + \mathcal{K}(|P|)$, as this value is the farthest a seed can be expanded to the left while its transcript cost remains not more than $\mathcal{K}(|P|)$.

Similar to head_occs, cands indirectly describes the candidates. That is, it stores the seeds (seeds), and how to expand them (add_cost) so as to produce the

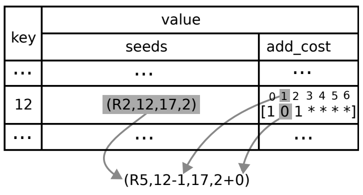


Fig. 9: An excerpt of the `cands` hash table, containing the candidates for $R3$, $R5$, and $R7$.

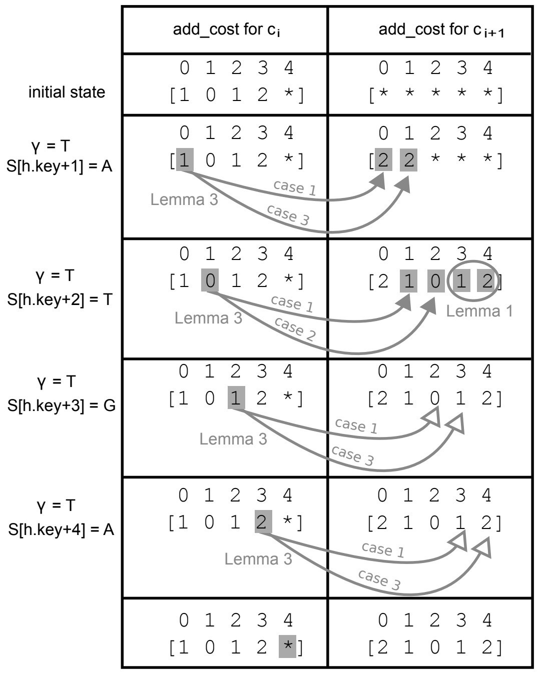


Fig. 10: Method `pExp` updates the `add_cost` array of `head_occs` entry h.

candidates. We illustrate the above using the example lattice of Figure 4. Assume `cands` after the first update, containing the candidates for regions $R3$, $R5$, and $R7$. Figure 9 depicts the entry `cands[12]`. Observe that seed $(R2, 12, 17, 2)$ produces the single candidate $(R5, 11, 17, 2)$, as explained in the figure, since all other occurrences exceed the cost threshold.

3.3.2 Phase 1: Producing the core occurrences

The goal of phase 1 is to produce all core occurrences with cost at most $\mathcal{K}(|P|)$. Note that an off-the-shelf ASM algorithm (like those discussed in Section 5) *cannot* produce all core occurrences. This happens because, by design, all ASM algorithms ignore some of the overlapping occurrences. For instance, if two occurrences have the same end position but different transcript costs, only the one with the lowest cost is reported. To illustrate this assume the classic dynamic programming algorithm described in [9]. Consider the core occurrence $(R1, 12, 16, 2)$ in Figure 5. It would not be among the results, because $(R1, 13, 16, 1)$ and $(R1, 14, 16, 1)$ end at the same position but have better transcript cost. However, $(R1, 12, 16, 2)$ should not be discarded, as it produces (via expansions in the last two phases) $(R14, 8, 17, 2)$, which is an ARSM result.

In order to produce all core occurrences, we follow three steps: (1) we execute a conventional ASM algorithm to discover the endpoints of the occurrences, (2) we construct disjoint windows around these endpoints, so that any occurrence is completely located within a window, and (3) in each window, we execute a variation of the dynamic programming algorithm (described below) to efficiently produce all occurrences.

Note that any ASM algorithm, including index-based solutions, can be used in step 1. Subsequently, in step 2, we construct the window $[i - |C| - \mathcal{K}(|P|) + 1, i]$ for each endpoint i found at the previous step. This is because it is impossible for an occurrence to start before the $(i-|C|-\mathcal{K}(|P|)+1)$-th position. To avoid redundant computations, we merge any overlapping windows.

Finally, in step 3, we execute a variation of the dynamic programming algorithm in [31] that is specifically adapted to return all occurrences. Recall that in conventional algorithms, each dynamic programming cell $DP[i, j]$ contains the cost and the start position of the best occurrence of the pattern prefix $P_{[1,i]}$ ending at position j in the data sequence S. On the other hand, in our variant, we keep in each cell $DP[i, j]$ an array which contains the costs of the best occurrences of $P_{[1,i]}$ ending at position j *for each possible start position* in S. Finally, since conventional algorithms cannot retrieve occurrences whose transcripts start with Insert operations, we must take special care so as not to miss them.

3.3.3 Phase 2: Producing the seeds

The second phase of `PS-ARSM` produces the seeds of all suffix chains, by progressively prefix-expanding head occurrences. First, `head_occs` is initialized with the core occurrences found in Phase 1. Then, `PS-ARSM` proceeds in sc_num-1 iterations. In each iteration, `PS-ARSM` executes two tasks: (1) it invokes `pExp` to update `head_occs` so as to contain the head occurrences of the next suffix chain, and (2) it invokes `getSeeds` to identify the seeds and initialize the `cands` hash table, which is required for Phase 3.

We now describe the `pExp` method. Assume that `head_occs` contains the head occurrences of suffix chain c_i. Then, `pExp` updates `head_occs` (by modifying the `add_cost` arrays) so that it contains the head occurrences of c_{i+1}. In particular, `pExp` visits a hash entry of `head_occs` and applies the prefix expansion rules for each possible end position of an occurrence. Fix a `head_occs` entry h with key $h.key$, and let γ be the last symbol in the head (first region) of suffix chain c_{i+1}. Let `add_cost`(c_i) (resp. `add_cost`(c_{i+1})) denote the array for suffix chain c_i (resp. c_{i+1}). Initially `add_cost`(c_{i+1}) is filled with $*$ values. Then, `pExp` scans `add_cost`(c_i) and applies the following procedure for each entry until a $*$ value is encountered. Consider the j-th entry in `add_cost`(c_i). Recall that this represents head occurrences of c_i that end at position $h.key + j$. First, the update procedure applies Lemma 3 (cases 1, 2 if symbols γ and $S_{[h.key+j]}$ match, or cases 1, 3 otherwise) to compute the additional costs of head occurrences of c_{i+1} that end at positions $h.key+j$ and $h.key+j+1$. If the corresponding entries in `add_cost`(c_{i+1}) have higher additional costs, they are updated. Finally, `pExp` applies Lemma 1 for each entry of `add_cost`(c_{i+1}). As before, it only updates an entry when the computed additional cost is smaller than the entry's existing value.

Figure 10 illustrates an application of `pExp`, assuming the maximum allowed cost is 2. The left column shows the `add_cost` array for suffix chain c_i, while the right column shows how `add_cost` array for the next suffix chain c_{i+1} is updated. The first row shows that `add_cost`(c_{i+1}) has initially all $*$ values. Then, `pExp` examines entry `add_cost`$(c_i)[0]$, and applies cases 1 and 3 of Lemma 3 ($S_{[h.key+1]} = \texttt{A} \neq \texttt{T} = \gamma$). This implies that entries 0 and 1 of `add_cost`(c_{i+1}) have cost 1 more than that in `add_cost`$(c_i)[0]$. Note that Lemma 1 does not update any `add_cost`(c_{i+1}) entry as it would get additional cost larger that the maximum allowed.

Next, `pExp` examines entry `add_cost`$(c_i)[1]$ (see third row of Figure 10). This time, cases 1 and 2 of Lemma 3 apply. Therefore, entry 1 of `add_cost`(c_{i+1}) is updated with additional cost $0 + 1 = 1$ (case 1), as it is lower than its current value. On the other hand, entry 2 of `add_cost`(c_{i+1}) has additional cost $0 + 0 = 0$ (case 2). Lemma 1 fills each remaining entry with additional costs 1 more compared to the previous entry.

Cases 1 and 3 of Lemma 3 apply for `add_cost`$(c_i)[2]$. However, since they produce occurrences with costs worse than those produced in the previous step, `add_cost`(c_{i+1}) is not updated (see fourth row of Figure 10). The `pExp` method continues with the next entries and terminates when it reaches $*$ in the final entry.

Finally, we describe the `getSeeds` method. Its goal is to identify the seeds among the head occurrences of the current suffix chain. Using the information in the *core_occs* and *add_cost* fields, `getSeeds` selects those head occurrences with cost not more than the allowable for the current suffix chain. The selected occurrences are the seeds, and are inserted in the `cands` hash table.

3.3.4 Phase 3: Producing the ARSM results

The third phase of `PS-ARSM` produces the actual ARSM results, by progressively suffix-expanding the seeds of all suffix chains. Note that Phase 2 has initialized hash table `cands` with all seeds. Then, `PS-ARSM` proceeds in *sc_length* iterations. In each iteration, `PS-ARSM` executes two tasks: (1) it invokes `sExp` (except in the first iteration) to update `cands` so as to contain the candidates of the next region of all suffix chains, and (2) it invokes `sieve` and `insert` to produce the actual ARSM results.

The `sExp` method is similar to `pExp`, except the following differences: (1) it operates on hash table `cands`, (2) it applies Lemmas 4 and 2, and (3) it examines the data sequence backwards. In the interest of space, we do not detail its operations.

The `sieve` method identifies ARSM results among the current candidates in the hash table `cands`. Finally, `insert` adds these occurrences to the set of ARSM results, taking care so that the second requirement of Definition 1 is not violated.

3.4 Cost Analysis and Optimization

Cost of Phase 1. The first phase involves three steps. In the first two, a conventional ASM algorithm is used to mark windows of the data sequence S that contain core occurrences with cost not more than $\mathcal{K}(|P|)$. Assume that the dynamic programming algorithm with cut-off heuristic [31] is applied. Then, according to [3], the average processing time of the first two steps is at most $\left(\frac{\mathcal{K}(|P|)}{1-e/\sqrt{|\Sigma|}} + O(1)\right) \cdot |S| \cdot T_{DP} = O\left(\mathcal{K}(|P|) \cdot |S|\right)$, where T_{DP} is the required time to compute each dynamic programming cell (constant for a given system configuration).

In the third step of Phase 1, our dynamic programming variation (see Section 3.3.2) is executed for each window marked in the previous steps. In our analysis, we assume the worst case scenario, where the entire data sequence is marked as a single window. Following again the analysis in [3], this step requires $O(\mathcal{K}(|P|)^2 \cdot |S|)$ time on average, because each dynamic programming cell contains $2 \cdot \mathcal{K}(|P|) + 1$ values.

Putting everything together, Phase 1 has a total processing time of $O(\mathcal{K}(|P|)^2 \cdot |S|)$.

Cost of Phases 2 and 3. To determine the cost of the last two phases, observe that each core occurrence, which is computed in the first phase, will undergo the same number of (prefix and suffix) expansions. In particular, the number of prefix expansions is one less than the number of suffix chains, i.e., $|P| - \beta$. On the other hand, the number of suffix expansions is one less that the length of a suffix chain, i.e., $\alpha - 1$. In total, a core occurrence will undergo $|P| - \beta + \alpha - 1 = |P| - |C|$ expansions. Furthermore, the total number of core occurrences produced in Phase 1 is given by $|S| \cdot f(|C|, \mathcal{K}(|P|))$, where $f(|C|, \mathcal{K}(|P|))$ is the probability of a random sequence of length $|C|$ matching in a given position of the data sequence with transcript cost not more than $\mathcal{K}(|P|)$.

Putting everything together, and assuming that each expansion requires T_{EX} time (constant for a given system configuration), the processing time of Phases 2 and 3 is $|S| \cdot f(|C|, \mathcal{K}(|P|)) \cdot (|P| - |C|) \cdot T_{EX} = O(|S| \cdot f(|C|, \mathcal{K}(|P|)) \cdot (|P| - |C|))$.

Note that computing a closed formula for the function $f()$ is a difficult task [3]. However, it is possible to derive the following upper bound for $f(|C|, \mathcal{K}(|P|))$ based on the analysis in the appendix of [3]:

$$\sum_{i=|C|-\mathcal{K}(|P|)}^{|C|} \frac{1}{|\Sigma|^{|C|-\mathcal{K}(|P|)}} \binom{|C|}{|C| - \mathcal{K}(|P|)} \binom{i}{|C| - \mathcal{K}(|P|)} + \sum_{i=|C|+1}^{|C|+\mathcal{K}(|P|)} \frac{1}{|\Sigma|^{i-\mathcal{K}(|P|)}} \binom{|C|}{i - \mathcal{K}(|P|)} \binom{i}{i - \mathcal{K}(|P|)}.$$

Optimization. In some ARSM instances (e.g., when the ratio $\mathcal{K}(|P|)/|R|$ is large), it is possible that the number of core occurrences is so high that the total execution time of `PS-ARSM` is dominated by Phases 2 and 3. For such instances, it is often preferable to divide the problem into two ARSM sub-problems, where each has much fewer core occurrences than the original, and solve them independently. This can be achieved by splitting the lattice in two, and appropriately defining the pattern sequence and the core for each sub-problem — the threshold function $\mathcal{K}$ is common. Figure 11 shows the two ways (vertical and horizontal partition) to view an ARSM problem as two independent sub-problems. The core and pattern regions for each sub-problem are also illustrated. Note that at the end, the occurrences from one sub-problem must be checked against those of the other, so as to remove occurrences that violate the second requirement of Definition 1.

The only question that remains is when and how to perform this sub-problem division. Based on the analysis of the previous paragraphs we estimate the total execution time of three scenarios: (a) solving the original problem, (b) solving the two sub-problems produced by a vertical split of the lattice, and (c) solving the two sub-problems produced by a horizontal split of the lattice. The scenario having the smallest estimated cost is the one selected.

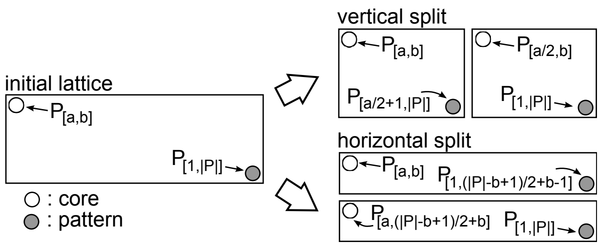


Fig. 11: Example of vertical and horizontal lattice split.

4 Experimental evaluation

We run a comprehensive set of experiments to assess the performance of our `PS-ARSM` method on both synthetic and real datasets. Section 4.1 describes the experimental setup and Section 4.2 presents our findings.

4.1 Setup

Algorithms. The evaluation involves three exact algorithms, i.e., they correctly retrieve all ARSM results.

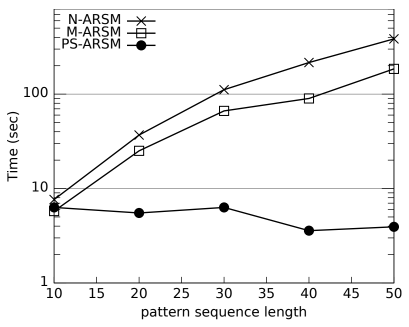
– `PS-ARSM`, our proposed solution for ARSM. The dynamic programming with the cut-off heuristic algorithm [31] is used in Phase 1.
– `N-ARSM`, the naïve approach that executes an ASM dynamic programming algorithm for every single region. The cut-off heuristic [31] is used to improve the dynamic programming efficiency.
– `M-ARSM`, which executes MASM [8] for each group of regions that have the same length. Experiments have shown this to be the most efficient MASM-based algorithm (see also Section 5).

We implement all algorithms in C++, and run the experiments on a dedicated Linux PC Intel Core 2 Duo CPU, E8400, at 3.00GHz. In our system, $T_{DP} = 3.14 \cdot 10^{-5} msec$ and $T_{EX} = 2.11 \cdot 10^{-5} msec$.

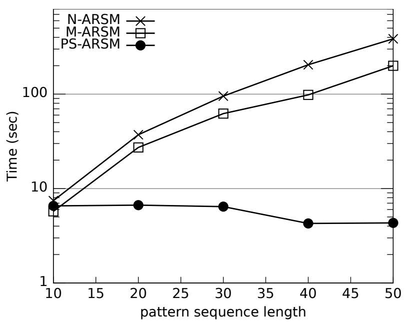
Parameters. We measure performance in terms of the total time required to produce the ARSM results, while we vary the following set of parameters: the pattern sequence length $|P|$ (measured in number of symbols); the data sequence length $|S|$; the ratio of the core to pattern

Table 2: Experimental parameters.

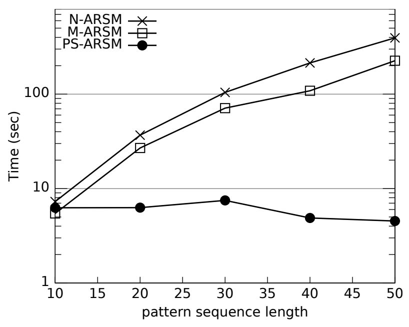
Parameter	Range of values	Default
$\|P\|$	10, 20, 30, 40, 50	30
$\|S\|$	1M, 5M, 10M, 50M, 100M	10M
$\|C\|/\|P\|$	0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8	0.5
$cPos$	left, middle, right	middle
α	0.1, 0.15, 0.2, 0.25, 0.3	0.2
$\|\Sigma\|$	4, 20, 94	4



(a) Synthetic dataset, D1



(b) 3'UTR dataset, D2



(c) CDS dataset, D3

Fig. 12: Varying the pattern sequence length $|P|$.

length $|core|/|P|$; the position of the core $cPos$; the ratio of the allowable transcript cost for each region over its length $\alpha = \mathcal{K}(|R|)/|R|$ (we consider linear threshold functions); the size of the alphabet $|\Sigma|$. Table 2 contains all parameters and their range of examined values. In each experiment, we vary a single parameter and set the remaining to the default values shown on the table.

Datasets. We use synthetic and real datasets. For the synthetic (D1), we use random sequences that follow the uniform Bernoulli model, i.e., each symbol has $1/|\Sigma|$ probability to occur and is selected independently of others. We generate 20 pairs of data and pattern sequences, and, for each pair, we execute the algorithms 5 times. Therefore, every reported time value is the average of 100 executions.

We also consider real datasets, obtained from the Ensembl database[1]. The 3'UTR dataset (D2) is a 44 million nucleotide sequence for the 3' untranslated region of the human gene transcripts. The CDS dataset (D3) is a 74 million nucleotide sequence for the coding region of the human gene transcripts. Note that these are genomic datasets and can only be used in experiments with alphabet size $\Sigma = 4$. We extract 20 random subsequences from these datasets to serve as the patterns, and for each of them we execute the algorithms 5 times. As a result, every reported time value is the average of 100 executions.

4.2 Results

Runtime analysis of `PS-ARSM`. We investigate the runtime performance of `PS-ARSM` for the default experimental setting. Note that, for these parameter values, `PS-ARSM` chooses to split the region lattice horizontally.

Table 3 presents the memory occupied by the two hash tables of `PS-ARSM`. The two runs shown correspond to the two executions of `PS-ARSM`, one for each half of the original lattice. Note that the size of `cands` is significantly smaller than that of `head_occs`. The reason is that much fewer occurrences survive after Phase 2. For example, as the table suggests for the first run on D1, there exist 128,577 distinct positions where occurrences end in Phase 2, but only 2,006 positions where occurrences start in Phase 3.

Table 3: Memory consumption of `PS-ARSM` hash tables.

Dataset	`head_occs` size	`cands` size
D1 run 1	7.69MB (128,577 entries)	0.17MB (2,006 entries)
D1 run 2	1.61MB (27,001 entries)	0.03MB (255 entries)
D2 run 1	7.82MB (126,852 entries)	0.37MB (3,221 entries)
D2 run 2	2.11MB (33,802 entries)	0.18MB (944 entries)
D3 run 1	9.51MB (155,912 entries)	0.33MB (3,898 entries)
D3 run 2	2.52MB (41,282 entries)	0.07MB (698 entries)

Table 4: Running time breakdown for `PS-ARSM` phases.

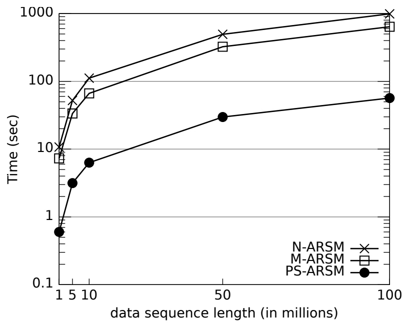
Dataset	Phase 1 (%)	Phase 2 (%)	Phase 3 (%)
D1	97.26	2.57	0.17
D2	96.70	2.88	0.42
D3	96.58	3.20	0.22

Table 4 presents the relative time spent in each phase of `PS-ARSM`; the values are based on the total execution time for both runs. Phase 1 is by far the most expensive as it consumes around 97% of the total running time. Among the other two phases, Phase 3 requires more time as it expands much fewer occurrences than Phase 2 (see Table 3).

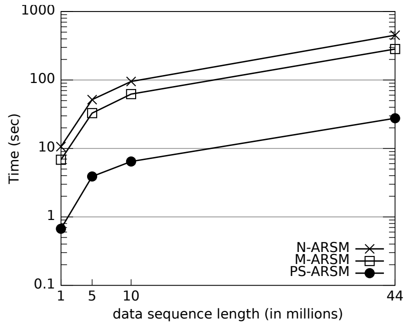
Varying the pattern sequence length. Figure 12 presents the execution times (in logarithmic scale) of all algorithms as the length of the pattern sequence $|P|$ varies. The findings are similar for all datasets. As the $|P|$ grows, the number of regions and their average length increases. This explains why the execution time of `M-ARSM` and `N-ARSM` grows.

On the other hand, the execution time of `PS-ARSM` remains less than 10 seconds, unaffected by $|P|$. Note

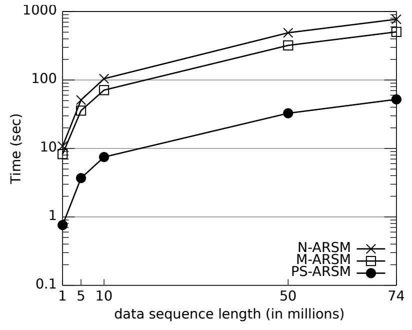
[1] http://www.ensembl.org/biomart/martview/



(a) Synthetic dataset, D1

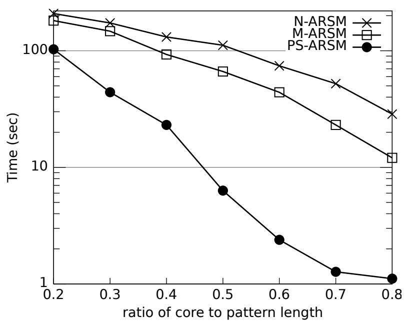


(b) 3'UTR dataset, D2

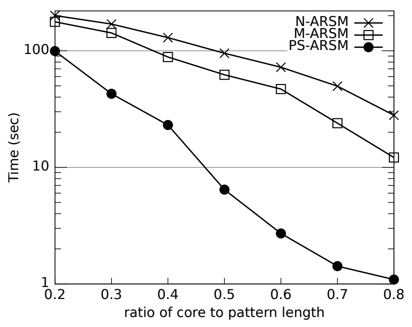


(c) CDS dataset, D3

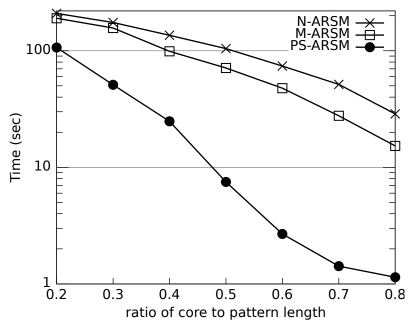
Fig. 13: Varying the data sequence length $|S|$.



(a) Synthetic dataset, D1



(b) 3'UTR dataset, D2



(c) CDS dataset, D3

Fig. 14: Varying the ratio of core to pattern length $|C|/|P|$.

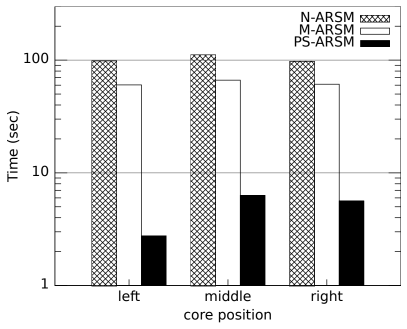
that as $|P|$ grows while the remaining parameters remain fixed, both the length of the core $|C|$ and the cost threshold $\mathcal{K}(|P|)$ also grow. In particular, the difference $|C| - \mathcal{K}(|P|) = 0.5 \cdot |P| - 0.2 \cdot |P| = 0.3 \cdot |P|$ (for the default values of Table 2) grows linearly with the pattern length. This difference plays a critical role in the number of core occurrences produced during the first phase of `PS-ARSM`. As the analysis of Section 3.4 shows, the number of core occurrences decreases rapidly as $|C| - \mathcal{K}(|P|)$ increases. Therefore, although the execution time of Phase 1 increases, that of Phases 2 and 3 decreases with $|P|$. As a result, for large $|P|$ values, `PS-ARSM` is almost two orders of magnitude faster.

Varying the data sequence length. Figure 13 shows the execution time on data sequences of different lengths. Note that to construct a sequence of length $|S|$ from the real datasets, we extract the first $|S|$ symbols, and that the maximum possible length is 44M for D2 and 74M for D3. The execution time of all methods grows linearly, as the data sequence size increases. Therefore, the performance improvement of `PS-ARSM` over `M-ARSM` and `N-ARSM`, in all datasets and $|S|$ values, is over one order of magnitude.

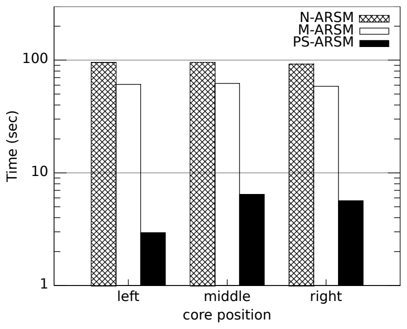
Varying the core/pattern length ratio. Figure 14 illustrates the execution time for several values of the $|C|/|P|$ ratio. The results are similar for all datasets. As the $|C|/|P|$ ratio increases, the execution time of all methods decreases. This is because the number of regions decreases and the average region length increases. Note that the benefit of `PS-ARSM` over its competitors increases with the ratio as, in addition to the above, the number of core occurrences produced in its first phase decreases. The reason is that $|C|$, and thus $|C|-\mathcal{K}(|P|)$, increases (see Section 3.4).

Varying the position of the core. Figure 15 presents the execution time as the location of the core in the pattern changes. As we move the core position to the start or to the end of the pattern (while the other parameters remain fixed) the number of regions decreases slightly. Therefore, all the algorithms run faster in this case. However, the decrease in the execution time of `PS-ARSM` is more pronounced. Briefly, the reason is that, in addition to the previous, the number of core occurrences are fewer when the core lies near the edges of the pattern. A detailed explanation when the core is at the left follows (the explanation for the other case is similar).

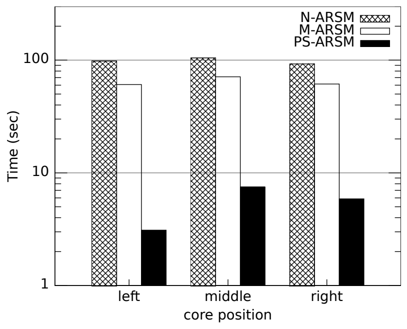
Consider the case of the core in the middle $C_m = [a_m, b_m]$ and that at the left of the pattern $C_l = [a_l, b_l] = [a_m - c, b_m - c]$, where $c > 0$. In both cases, `PS-ARSM` chooses to split the lattice horizontally. Let C_m^{top} and P_m^{top} (resp. C_m^{bot} and P_m^{bot}) denote the core and the pattern for the top (resp. bottom) half lattice when



(a) Synthetic dataset, D1

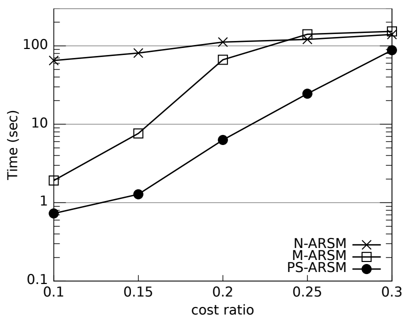


(b) 3'UTR dataset, D2

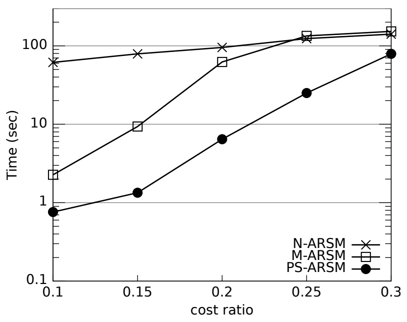


(c) CDS dataset, D3

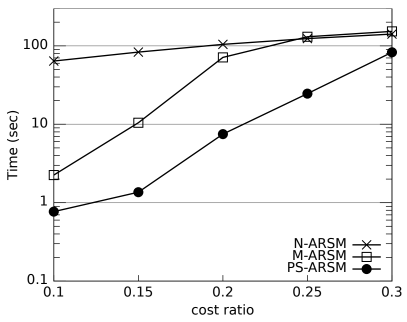
Fig. 15: Varying the core position $cPos$.



(a) Synthetic dataset, D1



(b) 3'UTR dataset, D2



(c) CDS dataset, D3

Fig. 16: Varying the cost ratio α.

the core lies in the middle. The corresponding notation for the core at the left is obtained by substituting m with l. Then, for the top half lattice, it holds that $|C_m^{top}| = |C_l^{top}|$ and $|P_l^{top}| = (|P_l| + b_l - 1)/2 = |P_m^{top}| - c/2 < |P_m^{top}|$. In other words, when the core is at the left, the top half pattern is smaller, and thus the allowable cost is also smaller, which leads to fewer core occurrences. On the other hand for the bottom half lattice, it holds that $|P_m^{bot}| = |P_l^{bot}|$, and $|C_l^{bot}| = (|P_l| + b_l + 1)/2 - a_l + 1 = |C_m^{bot}| + c/2 > |C_m^{bot}|$. Here, the bottom half pattern, and thus the allowable cost, is the same, but the bottom half core is larger, which again means fewer occurrences when the core is at the left. Overall, the total number of core occurrences in both lattices is smaller when the core is at the left.

Varying the cost ratio. Figure 16 shows the execution time as the cost threshold ratio varies. Higher α values require more effort by all methods. However, the performance of `M-ARSM` and `PS-ARSM` deteriorates faster and approaches that of the naïve method. Intuitively, the reason is the following. These two methods try to filter out certain areas of the data sequence that do not contain ARSM results. When the allowable transcript cost increases, fewer and smaller areas are excluded, and thus their filtering benefit diminishes and they behave similar to the brute force method of `N-ARSM`.

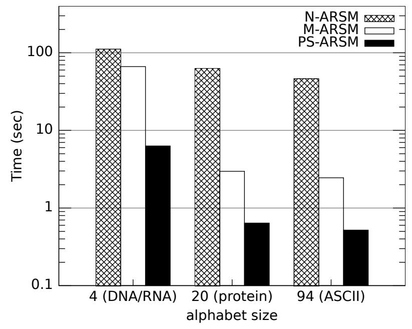


Fig. 17: Varying the alphabet size Σ.

Varying the alphabet size. Although the focus of this work is on genomic databases, where $|\Sigma| = 4$, we also examine the behavior of all methods on databases with different alphabet size. In particular, we consider protein sequences, where $|\Sigma| = 20$, and ASCII text sequences, where $|\Sigma| = 94$. In this experiment, we only use synthetic data. Figure 17 shows the results. The execution time of all methods decreases as $|\Sigma|$ increases, because their exist fewer possible occurrences (see Section 3.4). Note that both `M-ARSM` and `PS-ARSM` become significantly more efficient than the naïve method, while the benefit of `PS-ARSM` over `M-ARSM` remains close to one order of magnitude.

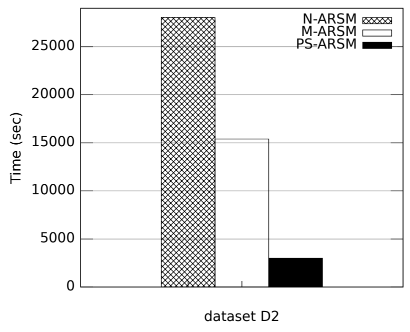


Fig. 18: Setting with real patterns.

Using real patterns. In the final experiment, we investigate the performance of PS-ARSM for the real-life problem of predicting micro-RNA bindings. Based on several biological observations (e.g., [7]), we select the following parameters. We use the 3'UTR dataset (D2) as the data sequence, since most known micro-RNA bindings are located in this gene section. Further, we randomly select 100 micro-RNA sequences from MirBase[2] as the patterns. Since the most important symbols for the binding are located at the left of the micro-RNA sequences, we select $|C| = 10$, $cPos = left$, and set $\alpha = 0.2$ for all regions, except the core where the allowable transcript cost is restricted to 1.

Figure 18 presents the results. Note that we report the total running times for all 100 micro-RNA patterns. The findings are similar to the case of synthetic patterns. PS-ARSM is around one order of magnitude faster than N-ARSM, and more than five times faster than M-ARSM.

5 Related Work

ARSM is a novel problem that belongs to the broad family of *inexact sequence matching* problems. References of such problems are encountered in the sixties and seventies, in a number of different fields like signal processing, text retrieval and computational biology [9,27]. Generally speaking, inexact sequence matching searches for sequences that are "similar" to a given pattern sequence. There exist various metrics to quantify sequence similarity, such as *similarity measures* (e.g., SW score [30]) or *distance/deviation measures* (e.g., edit distance [14,15]).

The most basic inexact sequence matching problem is *global alignment* [24,28]. Its goal is to compute the edit distance between two sequences, and thus determine the transcript with the minimum cost. Intuitively, based on this transcript, the two strings are globally (i.e., completely) aligned opposite each other with the minimum number of spaces and mismatches. A dynamic programming (DP) algorithm [28] can compute the edit distance in $O(nm)$ time, and determine the optimal transcript by backtracking the DP array in $O(n + m)$ time, where n, m are the lengths of the sequences.

Given a pattern and a data sequence, the objective of the *semi-global* (also known as global-local) *alignment* [29] is to find a *subsequence* of the data that has the smallest edit distance to the pattern. Intuitively, the pattern is aligned only to this subsequence, and not to the entire data sequence. An important variation is to retrieve *all* data subsequences that have edit distance below a specified threshold k [21]. Throughout this paper, we refer to this problem as *Approximate Sequence Matching*, and denote it as ASM. There exist several works surveying proposed algorithms for this problem, e.g., [11,21]. The most known solution is the dynamic programming of Sellers [29], which has running time $O(nm)$, where n, m are the lengths of the data and pattern sequence, respectively.

Several improvements to the basic DP formulation for ASM have been proposed; see e.g., [21]. One such optimization is the *cut-off heuristic* [31], which we use in our DP implementations. The basic idea is that, given an edit distance threshold k, all dynamic programming cells that have value greater than k are not required when reporting the results. Therefore, upon filling a DP column, the algorithm determines which cells in the next column need not be computed, and avoid unnecessary computations. It is shown that the cut-off heuristic has an expected running time of $O(kn)$.

The most efficient algorithms for ASM belong to the group of *filtering algorithms*. Chang's [6] and Fredriksson's [8] algorithms have optimal average case time complexity [6]. These algorithms first compare the pattern to any sequence of a predetermined length ℓ, called ℓ-gram, and then use this information to filter out areas of the data sequence that cannot contain an ASM result. The remaining areas are processed using a conventional ASM algorithm.

Note that an ARSM instance can be stated as several ASM instances, one for each region of the pattern. Therefore, any ASM algorithm can solve the ARSM problem. The drawback, however, is that this approach performs a large number of redundant computations, mainly because the regions are overlapping sequences.

A more attractive approach is to use algorithms designed for the *Multiple ASM (MASM)* problem. The goal of MASM is to retrieve the ASM results for a group of overlapping patterns. While several methods exist

[2] http://www.mirbase.org/, the database where all micro-RNA sequences are registered.

(e.g., [20,4,10,8]), Fredriksson's algorithm [8] is proven to be optimal [23]. This algorithm scans the data sequence using a sliding window. For each window position, it reads backwards (i.e., from the right to the left) consecutive, non-overlapping ℓ-grams. When the aggregated deviation of the read ℓ-grams exceeds a threshold, the algorithm skips the current window and slides it to the right. Otherwise, it must examine the window for results.

Although it is possible to directly apply a MASM algorithm for the ARSM problem, it is not recommended for two reasons. First, it is not efficient to select the same ℓ value for all ARSM regions, as the optimal choice depends on the region length and its maximum allowed deviation. Second, MASM algorithms are designed for pattern of similar length, whereas the length of ARSM regions may vary considerably. Based on these observations, the most efficient approach would be to group regions according to their length, and execute a MASM algorithm once per group. Still, this method cannot exploit overlaps in regions *across* groups.

An important inexact sequence matching problem is *local alignment* [30]. Given a pattern P and a data sequence S, the goal is to determine two subsequences, one from P and one from S, that have the highest similarity score. For an important class of similarity scores, the dynamic programming of Smith-Waterman [30] can solve this problem in $O(nm)$ time, where $n = |S|$ and $m = |P|$. When searching for highly similar subsequences, the high computational cost of DP algorithms makes approximation solutions (e.g., [17,26, 1,2]) more attractive. These methods use heuristics to avoid searching parts of the sequences that are unlikely to contain a local alignment. As a side-effect they may miss results. The most known approximation solution is BLAST [1] and its variations [2,32,13,12].

Local alignment methods are not suitable for the ARSM problem. Even exact DP-based algorithms, e.g., Smith-Waterman (SW) [30], are not guaranteed to return all ARSM answers. Consider the data sequence $S = \cdots \texttt{GTTGA} \cdots$, and the region $R = \texttt{GCCGA}$. Clearly, there exists an occurrence of R in S with cost (edit distance) 2. However, SW would fail to identify it[3]. The reason is the following. In the DP array, the cell corresponding to the two `A`s in the sequences has the highest value 2, meaning that there exists two subsequences ending in `A` that have similarity score 2. However, since this score corresponds to the `GA` subsequences, there is no way to backtrack and identify the `GCCGA`, `GTTGA` subsequences.

[3] We assume that a match has score 1, whereas a mismatch, delete or insert has score −1. Note that a similar counter-example exists for other scores.

Several index structures, e.g., suffix trees, suffix arrays, q-grams, etc., can be applied to inexact sequence matching problems (see [22]). For instance, the methods in [5,19,25,16] utilize indices to quickly eliminate parts of the search space and focus on areas that may contain results. Note that any such ASM algorithm, e.g., [25, 5], can be applied in Phase 1 of `PS-ARSM`.

6 Conclusion

We introduce a novel approximate sequence matching problem, the ARSM problem. Its objective is to retrieve all regional occurrences of a pattern in a data sequence. The matching regions of the pattern must contain a predetermined area of the pattern, the core. Moreover, the allowable deviation from the data sequence is stricter for smaller and looser for larger regions.

To deal with the previous problem, we propose the `PS-ARSM` method. Our method takes advantage of the prefix and suffix overlaps avoiding redundant computations. A detailed experimental evaluation shows that `PS-ARSM` is up to two orders of magnitude faster than existing techniques adapted to the ARSM problem.

References

1. Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403 – 410, October 1990.
2. Stephen F. Altschul, Thomas L. Madden, Alejandro A. Schaffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman. Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, September 1997.
3. Ricardo A. Baeza-Yates and Gonzalo Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.
4. Ricardo A. Baeza-Yates and Gonzalo Navarro. New and faster filters for multiple approximate string matching. *Random Structure and Algorithms*, 20(1):23 – 49, 2002.
5. Ricardo A. Baeza-Yates and Chris H. Perleberg. Fast and practical approximate string matching. *Inf. Process. Lett.*, 59(1):21–27, 1996.
6. William I. Chang and Thomas G. Marr. Approximate string matching and local similarity. In *Combinatorial Pattern Matching (CPM)*, volume 807 of *Lecture Notes in Computer Science (LNCS)*, pages 259–273. Springer, 1994.
7. John G. Doench and Phillip A. Sharp. Specificity of microrna target selection in translational repression. *Genes Dev.*, 18(5):504–511, 2004.
8. Kimmo Fredriksson and Gonzalo Navarro. Average-optimal single and multiple approximate string matching. *ACM Journal of Experimental Algorithms*, 9, 2004.
9. Dan Gusfield. *Algorithms on strings, trees, and sequences.* Cambridge University Press, 1999.
10. Heikki Hyyrö and Gonzalo Navarro. Faster bit-parallel approximate string matching. In *Combinatorial Pattern Matching (CPM)*, volume 2373 of *Lecture Notes in Computer Science (LNCS)*, pages 203–224. Springer, 2002.

11. Petteri Jokinen, Jorma Tarhio, and Esko Ukkonen. A comparison of approximate string matching algorithms. *Softw., Pract. Exper.*, 26(12):1439–1458, 1996.
12. You Jung Kim, Andrew Boyd, Brian D Athey, and Jignesh M Patel. miblast: scalable evaluation of a batch of nucleotide sequence queries with blast. *Nucleic Acids Research*, 33(13):4335–44, 2005.
13. I. Korf and W. Gish. Mpblast: Improved blast performance with multiplexed queries. *Bioinformatics*, 16(11):1052–1053, November 2000.
14. Vladimir Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Probl. Inf. Transmission*, 1:8 – 17, 1965.
15. Vladimir Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Sov. Phys. Dokl.*, 10(8):707 – 710, 1966. Original in Russian in Dokl. Akad. Nauk SSSR 163(4): 845-848, 1965.
16. Yinan Li, Allison Terrell, and Jignesh M. Patel. Wham: a high-throughput sequence alignment method. In *SIGMOD Conference*, pages 445–456, 2011.
17. D. J. Lipman and W. R. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435 – 1441, March 1985.
18. M. Maragkakis, M. Reczko, V. A. Simossis, P. Alexiou, G. L. Papadopoulos, T. Dalamagas, G. Giannopoulos, G. Goumas, E. Koukis, K. Kourtis, T. Vergoulis, N. Koziris, T. Sellis, P. Tsanakas, and A. G. Hatzigeorgiou. Diana-microt web server: elucidating microrna functions through target prediction. *Nucleic Acids Research*, 37(suppl 2):W273–W276, 2009.
19. Colin Meek, Jignesh M. Patel, and Shruti Kasetty. Oasis: An online and accurate technique for local-alignment searches on biological sequences. In *VLDB*, pages 910–921, 2003.
20. R. Muth and U. Mamber. Approximate multiple string search. In *Combinatorial Pattern Matching (CPM)*, volume 1075 of *Lecture Notes in Computer Science (LNCS)*, pages 75–86. Springer, 1996.
21. Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys (CSUR)*, 33(1):31 – 88, March 2001.
22. Gonzalo Navarro, Ricardo A. Baeza-Yates, Erkki Sutinen, and Jorma Tarhio. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin (DEBU)*, 24(4):19 – 27, December 2001.
23. Gonzalo Navarro and Kimmo Fredriksson. Average complexity of exact and approximate multiple string matching. *Theor. Comput. Sci.*, 321(2-3):283 – 290, 2004.
24. S B Needleman and C D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–53, March 1970.
25. Panagiotis Papapetrou, Vassilis Athitsos, George Kollios, and Dimitrios Gunopulos. Reference-based alignment in large sequence databases. *PVLDB*, 2(1):205–216, 2009.
26. W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proc Natl Acad Sci USA*, 85(8):2444 – 2448, April 1988.
27. David Sankoff and Joseph Kruskal. *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison.* Addison-Wesley,Reading,MA, 1983.
28. Peter H. Sellers. An algorithm for the distance between two finite sequences. *J. Combin. Theory Ser. A*, 16:253–258, 1974.
29. Peter H. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *Journal of Algorithms*, 1(4):359 – 373, 1980.
30. T F Smith and M S Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–7, March 1981.
31. Esko Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6:132–137, 1985.
32. Z. Zhang, S. Schwartz, L. Wagner, and W. Miller. A greedy algorithm for aligning dna sequences. *J. Comput. Biol.*, 7(1-2):203–214, 2000.