Quick-Motif: An Efficient and Scalable Framework for Exact Motif Discovery

Yuhong Li #1, Leong Hou U #2, Man Lung Yiu *3, Zhiguo Gong #4

#Department of Computer and Information Science, University of Macau
Av. Padre Tomás Pereira Taipa, Macau
1yb27407@umac.mo 2ryanlhu@umac.mo 4fstzgg@umac.mo

*Department of Computing, Hong Kong Polytechnic University
Hung Hom, Kowloon, Hong Kong
3csmlyiu@comp.polyu.edu.hk

***Abstract*—Discovering motifs in sequence databases has been receiving abundant attentions from both database and data mining communities, where the motif is the most correlated pair of subsequences in a sequence object. Motif discovery is expensive for emerging applications which may have very long sequences (e.g., million observations per sequence) or the queries arrive rapidly (e.g., per 10 seconds). Prior works cannot offer fast correlation computations and prune subsequence pairs at the same time, as these two techniques require different orderings on examining subsequence pairs. In this work, we propose a novel framework named Quick-Motif which adopts a two-level approach to enable batch pruning at the outer level and enable fast correlation calculation at the inner level. We further propose two optimization techniques for the outer and the inner level. In our experimental study, our method is up to 3 orders of magnitude faster than the state-of-the-art methods.**

I. Introduction

The motif discovery problem has been shown to have great utility for several data mining algorithms, including clustering, classification, sequence summarization, and rule discovery [1], [2], [3], [4], [5]. Given a sequence object (representing the data), this problem reports the *motif* as the most correlated pair of subsequences in a sequence object. The correlation between subsequences is measured by a correlation metric, e.g., Pearson correlation. As an example, Figure 1 illustrates a weekly motif discovered in a power consumption dataset [6].

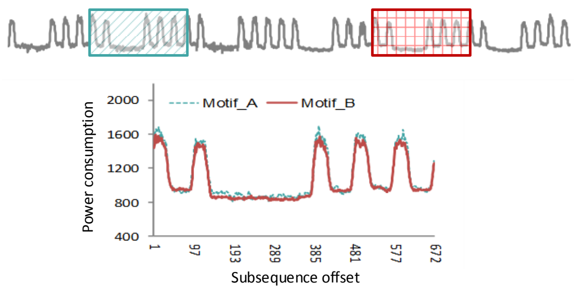


Fig. 1: Weekly motif discovered in a sequence (35,000 values) that records the average power consumption for a Dutch research facility in the year 1997

This problem has received significant attention from both databases and data mining communities [1], [2], [3], [4], [5]. Discovering motif is a core subroutine for activity discovery of humans and animals, with applications in elder care, surveillance and sports training [3]. Besides, clustering enumerated motifs is shown to be more meaningful than clustering all the subsequences in a long sequence [7]. As in other domains, this approximately repeated structure may be conserved for some reason that is of interest to domain specialists [4].

It is time consuming to solve the motif discovery problem. Note that a sequence object of length m contains $m - \ell + 1$ subsequences of length ℓ. The brute force method would (i) examine all pairs of subsequences (i.e., $O((m - \ell)^2)$ pairs) and then (ii) compute the correlation for each pair (in $O(\ell)$ time). This method takes $O((m - \ell)^2 \cdot \ell)$ time, which is too expensive for a long sequence. Most of the literature focuses on fast algorithms for approximate motif discovery [1], [2], [8]; however, they do not provide guarantee on the result quality.

Recently, Mueen et al. [5], [9] propose two efficient algorithms for exact motif discovery. The smart brute force method (SBF) [9] examines subsequence pairs in a specific ordering in order to compute the correlation of each pair incrementally in (amortized) constant time. However, this method always examines $O((m - \ell)^2)$ subsequence pairs as it cannot prune any subsequence pair. On the other hand, the reference indices method (MK) [5] examines subsequences by the order of correlation to a reference, and employs a pruning technique to discard unpromising subsequence pairs. Then, it computes the correlation for each remaining pair (in $O(\ell)$ time). Nevertheless, its pruning effectiveness relies heavily on the data distribution. In the worst case, the number of remaining pairs can be $O((m-\ell)^2)$ and it degrades to the brute force method. In our experimental study, when compared to the brute force method, MK computes exact distances for only 0.013% of all pairs on the ECG dataset [10], but 89.37% of all pairs on the EPG dataset[1]. Also, MK requires considerable memory space for storing all normalized subsequences and reference indices. We summarize the time and space complexity of existing methods in Table I. We do not show the construction cost as they are small when compared to their overall cost (cf. Section IV-A).

It is tempting to ask whether we can obtain a more efficient algorithm by combining the incremental correlation computation technique (in SBF) with the pruning technique (in MK).

[1]http://www.cs.ucr.edu/ mueen/txt/insect15.txt

TABLE I: Time and space complexities of exact methods

Methods	Time Complexity	Space
Brute force	$O((m-\ell)^2 \cdot \ell)$	$O((m-\ell) \cdot \ell)$
SBF [9]	$O((m-\ell)^2)$	$O(m)$
MK [5]	$O((m-\ell)^2 \cdot (1-P_{lb}) \cdot R$ + $(m-\ell)^2 \cdot (1-P_r) \cdot \ell)$	$O((m-\ell) \cdot (R+\ell))$
QM	$O((\frac{m-\ell}{w})^2 \cdot (1-P_{lb}) \cdot \phi$ + $(m-\ell)^2 \cdot (1-P_r) \cdot \frac{\ell}{w})$	$O(m+N_{survive})$

R: number of references in MK
w: grouping size in QM
ϕ: PAA dimensionality in QM
P_{lb}: the pruning probability of lower bound computation
P_r: the pruning probability of real distance computation
$N_{survive}$: number of surviving group pairs in QM

Unfortunately, since SBF and MK rely on different orderings on examining subsequence pairs, these two techniques cannot be readily combined together. To the best of our knowledge, no prior work jointly applies both techniques to discover motif.

In this paper, we propose a two-level approach that enables both techniques together. The idea is to group subsequences by their offset locality, say, assigning every w consecutive subsequences to the same group. First, at the outer level, we examine all pairs of groups and prune unpromising group-pairs. Second, for each remaining group-pairs, we can apply the incremental correlation computation technique for all subsequence pairs within the group (i.e., at the inner level). Furthermore, we propose two optimization techniques for both the outer and the inner levels: (i) a locality-based searching strategy for discovering the *true* motif as soon as possible, and (ii) a batch refinement technique that shares the processing cost of surviving group-pairs (i.e., promising pairs).

The improvement of our work is brought by Lemmas 1, 2, and 3. As shown in Table I, our proposed method (QM) achieves a lower worst-case time complexity than MK [2]. In practice, our method is up to 3 orders of magnitude faster than prior solutions. For example, QM only takes 0.88s to discover a motif of length 900 in TAO dataset, however MK and SBF will need 798s and 1257s for the same task respectively. This dramatic performance gain makes online processing feasible for motif discovery in very long sequences. Besides, our method is ready for parallel implementation which performs well with multi-core or distributed systems.

The remaining structure of our paper is organized as follows. In Section II, we formally define the motif discovery problem and briefly discuss existing solutions for this problem. The framework Quick-Motif is proposed and analyzed in Section III. We experimentally evaluate our framework in Section IV. The related work is summarized in Section V and we conclude our work in Section VI.

II. Preliminaries

In this section, we formally define the motif discovery problem and briefly discuss the latest solutions for this problem.

[2]It should be noted that the pruning ratios, P_{lb} and P_r, are more sensitive to the data than the methodology.

A. Motif Discovery

We first define the notation of sequence object s and its subsequences as follows. These notations are thoroughly used in this work.

Definition 1 (Subsequence of s): Given a sequence object s of length m (i.e., $s = s[0]...s[m-1]$), a valid subsequence of length ℓ in s is denoted as $s_i = s[i]...s[i+\ell-1]$, where i is the offset of the subsequence and $0 \leq i < m - \ell + 1$.

According to the definition, a motif of length ℓ in a sequence object s means the most correlated subsequence pair (s_i, s_j) [3]. In this work, we use normalized Euclidean distance as the underlying distance measure which is commonly used in motif discovery studies [3], [4], [5], [9]. Note that other distance measures (e.g., p-norm or DTW) can be used to measure the distances but the lack of normalization renders them unsuitable for subsequences with different offset and scale [3], [11], [12]. To produce meaningful motif result, we omit overlapping subsequence pair (e.g., s_i and s_{i+1}) as they are *trivially matched* [13], which are not interesting for data analysts to explore further. Thus, in this work we report a subsequence pair as a motif only if the pair is non-trivial (i.e., non-overlapping). The formal definition of motif discovery is as follows:

Problem 1 (Motif Discovery): Given a sequence object s and the targeted motif length ℓ, the motif discovery is to return a pair of subsequences (s_i, s_j), where the normalized Euclidean distance of s_i and s_j is minimum among all non-trivial subsequence pairs.

Generally the normalized Euclidean distance of a subsequence pair is calculated by the Euclidean distance of their normalized form (Z-normalization) [14], where the normalized form of a subsequence is defined as follows.

$$\hat{s}_i[k] = \frac{s_i[k] - \mu(s_i)}{\sigma(s_i)}, \forall_{0 \leq k < \ell} \tag{1}$$

where $\mu(s_i)$ and $\sigma(s_i)$ denote the mean and standard deviation of s_i, respectively. $\hat{s}_i$ indicates the normalized version of s_i. Accordingly, the normalized Euclidean distance can be represented by $d(\hat{s}_i, \hat{s}_j)$. In the sequel, $d(\hat{s}_i, \hat{s}_j)$ can be represented by their Pearson correlation [11] as

$$d(\hat{s}_i, \hat{s}_j) = \sqrt{2\ell(1 - \rho(s_i, s_j))} \tag{2}$$

where $\rho(s_i, s_j)$ is the Pearson correlation of s_i and s_j.

B. Preliminary: Running Sum Technique

The Pearson correlation for a pair (s_i, s_j) can be computed in $O(1)$ time, by utilizing running sums [9], [11]. We introduce this technique here as we will apply it in our solution. The Pearson correlation is defined as:

$$\rho(s_i, s_j) = \frac{\sum_{x=0}^{\ell-1} s[i+x]s[j+x] - \ell\mu(s_i)\mu(s_j)}{\ell\sigma(s_i)\sigma(s_j)} \tag{3}$$

[3]If not explicitly mentioned, the length of a subsequence s_i is ℓ in this work.

This technique requires keeping two running sum arrays as follows:

$$A[i] = \sum_{x=0}^{i} s[x], \quad A^2[i] = \sum_{x=0}^{i} s[x]^2, \quad 0 \le i < m \tag{4}$$

With these two running sum arrays, we can calculate the mean and standard deviation of a subsequence in $O(1)$ time.

$$\begin{aligned}\mu(s_i) &= \frac{1}{\ell}(\sum_{x=i}^{i+\ell-1} s[x]) \\ &= \frac{1}{\ell}(A[i+\ell-1] - A[i] + s[i]) \end{aligned}\tag{5}$$

$$\begin{aligned}\sigma(s_i)^2 &= \frac{1}{\ell}(\sum_{x=i}^{i+\ell-1} s[x]^2) - \mu(s_i)^2 \\ &= \frac{1}{\ell}(A^2[i+\ell-1] - A^2[i] + s[i]^2) - \mu(s_i)^2 \end{aligned}\tag{6}$$

To calculate the Pearson correlation in constant time, we need to calculate the *cross sum* (i.e., $\sum_{x=0}^{\ell-1} s[i+x]s[j+x]$) of Equation 3 in $O(1)$ time. The cross sum can be maintained in $O(1)$ time incrementally by Equation 7.

$$\begin{aligned}&\sum_{x=0}^{\ell-1} s[i+1+x]s[j+1+x] \\ =&(\sum_{x=0}^{\ell-1} s[i+x]s[j+x]) + s[i+\ell]s[j+\ell] - s[i]s[j]\end{aligned}\tag{7}$$

In summary, given $\sum_{x=0}^{\ell-1} s[i+x]s[j+x]$ and two running sum arrays (i.e., A and A^2), we can incrementally calculate $\rho(s_{i+1}, s_{j+1})$ in $O(1)$ time.

By simple induction, we can calculate the subsequence pairs of the same offset gap (i.e., $\rho(s_i, s_j)$, $\rho(s_{i+1}, s_{j+1})$, $\cdots$, $\rho(s_{i+\ell}, s_{j+\ell})$) incrementally. Thus, given m initial cross sums of the offset gaps, i.e., $\sum_{x=0}^{\ell-1} s[x]s[x]$, $\sum_{x=0}^{\ell-1} s[x]s[x+1]$, ..., $\sum_{x=0}^{\ell-1} s[x]s[x+m-\ell]$, the Pearson correlation of every subsequence pair can be calculated in $O(1)$ time.

C. Related Work

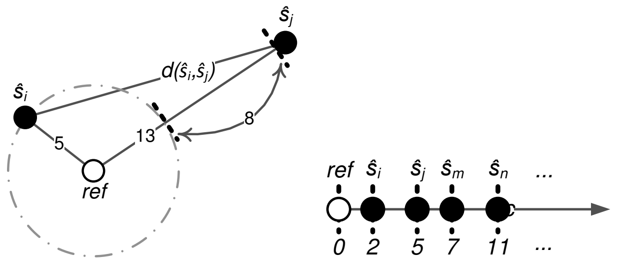
Brute force. We first introduce the brute force solution which compares every possible subsequence pair using standard distance calculation (e.g., distance early termination [5]). The pseudo code is shown in Algorithm 1. To efficiently calculate the normalized Euclidean distances, this naïve approach pre-normalizes all subsequences and keeps their normalized forms in main memory to avoid normalizing the same subsequence multiple times during the discovery process [4]. During the execution, the algorithm updates a best-so-far distance *bsf* when a better motif is discovered. The complexity of Algorithm 1 is $O((m-\ell)^2\ell)$, where $(m-\ell)^2$ is the number of all subsequence pairs and ℓ indicates the motif length.

Smart brute force (SBF). This solution [9] is to calculate the distance of the subsequence pairs in a specific order ($(s_i, s_j), (s_{i+1}, s_{j+1}), (s_{i+2}, s_{j+2}), \cdots$), which can reduce the time complexity of each distance calculation from $O(\ell)$ to $O(1)$ (line 5 in Algorithm 1). It applies the running sum technique in Section II-B.

Algorithm 1 Brute Force (BF)

```
   Input: sequence $s$ of length $m$, motif length $\ell$
   Output: motif offset $os$ and motif distance $bsf$
1: normalized all subsequences with length $\ell$ in sequence $s$
2: $bsf \leftarrow \infty$
3: for $i \leftarrow 0$ to $m - l$ do
4:     for $j \leftarrow i + \ell$ to $m - l$ do        ▷ use $\ell$ to avoid trivial match
5:         if $d(\hat{s}_i, \hat{s}_j) < bsf$ then
6:             $bsf \leftarrow d(\hat{s}_i, \hat{s}_j)$; $os \leftarrow (i, j)$
```

Reference indices (MK). Given a set of subsequences and their distances to a reference subsequence, we can derive the distance lower bound between any subsequence pair based on *triangular inequality*. Figure 2(a) illustrates this basic idea by an example. Suppose we have two subsequences $\hat{s}_i$ and $\hat{s}_j$ and their distance to a given reference ref, the lower bound of $d(\hat{s}_i, \hat{s}_j)$ can be calculated by $|d(ref, \hat{s}_i) - d(ref, \hat{s}_j)| = |5 - 13| = 8$ in this example. If the lower bound is already larger than *bsf* (i.e., the distance of the current motif candidate), then this subsequence pair can be pruned safely. The lower bound can be tightened if it is derived (as the maximum value) from multiple reference indices.



(a) pruning by reference indices, ℓ (b) execution strategy

Fig. 2: Techniques in MK

Based on the lower bounds derived from the reference indices, Mueen et al. [5] propose an efficient motif discovery solution, MK, which avoids examining every subsequence pair in the discovery process. Their approach first picks one reference to construct a sorted list based on the distance from each subsequence to the reference. This reference selection is based on the standard deviation of the distances. According to the order in the sorted list, MK progressively examines the subsequence pairs based on their offset gaps. Given the example in Figure 2(b), the first batch of subsequence pairs contains $(\hat{s}_i, \hat{s}_j)$, $(\hat{s}_j, \hat{s}_m)$, and $(\hat{s}_m, \hat{s}_n)$, where the offset gap of these pairs is 1. For each subsequence pair in the current batch, we estimate their lower bounds by the reference indices (i.e., taking $O(R)$ time from R reference indices) and calculate their real distances if necessary (i.e., taking $O(\ell)$ time). At the end of each batch, we process the next batch of subsequence pairs, e.g., $(\hat{s}_i, \hat{s}_m)$ and $(\hat{s}_j, \hat{s}_n)$, only if the lower bound provided by the sorted list of any pair in the current batch is better than *bsf*. Otherwise, we can safely terminate the process and report the best-so-far motif as the result since the *bsf* is already better than any unseen subsequence pair (due to the monotonicity of the distances with respect to the offset gaps).

The execution strategy of MK reduces the quadratic number of pairwise comparisons (lines 3-4 in Algorithm 1) to an acceptable level in practice. MK is recognized as the most efficient motif discovery solution in [9] and is used as a core component in other time series problems [1], [2], [7], [15]. However, the performance of MK is very sensitive to the dataset. Another obvious drawback of MK is the space overhead, which takes $O((R+\ell)(m-\ell))$ space [9] where $R(m-\ell)$ is the size of the reference indices and $\ell(m-\ell)$ is the space overhead for normalized subsequences (same as Algorithm 1) [4].

Discussion. Table I shows the time and space complexity of SBF and MK. The computational cost of MK contains two parts, $O((m-\ell)^2 \cdot (1-P_{lb}) \cdot R)$ for checking the distance lower bounds and $O((m-\ell)^2 \cdot (1-P_r) \cdot \ell)$ for computing the real distance of unpruned subsequence pairs. Regarding the worst-case time complexity, SBF is superior to MK since it not only reduces the time complexity from $O((m-\ell)^2 \cdot (\ell+R))$ to $O((m-\ell)^2)$ but also has lower space overhead, i.e., $O(m)$ (two running sum arrays). However, the strong pruning capability of MK makes it superior to SBF in practices. Their performances are evaluated thoroughly in our experimental section.

III. Quick-Motif

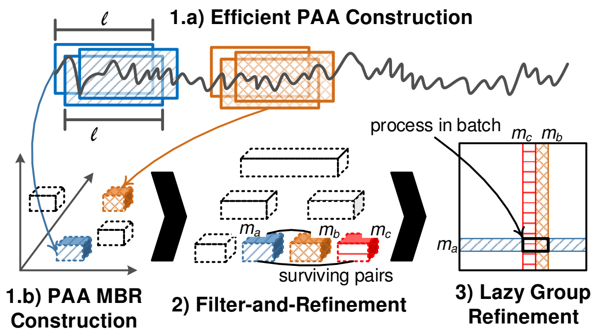


Fig. 3: The framework of Quick-Motif

To the best of our knowledge, prior solutions focus on one narrow aspect to boost the query performance. For instance, MK exploits the pruning capability using reference indices but it takes $O(\ell)$ time per distance calculation; SBF exploits the fast distance calculation in $O(1)$ time but without any pruning support. Thus, none of these solutions can offer acceptable performance for emerging applications when (i) the sequence is long, e.g., millions of values, or (ii) the motif discovery query is issued frequently, e.g., every minute. In this work, we exploit more broadly such that our solution, Quick-Motif, is more *smarter* than SBF as equipped with batch pruning capability and it can be used to answer the motif discovery problem at scale.

Figure 3 illustrates the work flow of our framework. To reduce the dimensionality of the problem, we first transform each subsequence into their PAA representation (Section III-A1). To support fast distance calculation, we group consecutive PAA representations into PAA MBRs such that the pairwise distances of two MBRs can be computed in $O(1)$ time (Section III-A2). To support batch pruning, we manage the MBRs into a Hilbert R-tree and apply a filter-and-refinement framework to prune unpromising MBR pairs (Section III-B). To further improve the performance, we propose a lazy group refinement technique which attempts to process surviving MBRs in one refinement batch (Section III-C). By taking the advantages of all these techniques, Quick-Motif outperforms the state-of-the-art approaches by up to 3 orders of magnitude in terms of response time, which addresses the need of emerging applications.

A. PAA MBR Construction

1) Efficient PAA Construction: PAA is an intuitive dimensionality reduction method, yet shown to be competitive with other dimensionality reduction representations like SVD, DFT and DWT as discussed in [16]. Specifically, a normalized subsequence $\hat{s}_i$ can be transformed into ϕ segments of equal length $\frac{\ell}{\phi}$. Formally, given a normalized subsequence $\hat{s}_i$, the k-th element (i.e., k-th line segment) of its ϕ-dimensional PAA representation is defined as follows.

$$e_{\hat{s}_i}[k] = \frac{\phi}{\ell} \sum_{x=\frac{\ell}{\phi}\cdot k}^{\frac{\ell}{\phi}(k+1)-1} \hat{s}_i[x] \tag{8}$$

To transform a subsequence s_i into its normalized PAA representation $e_{\hat{s}_i}$, a straightforward solution first normalizes s_i into $\hat{s}_i$ and then transforms $\hat{s}_i$ into $e_{\hat{s}_i}$ using Equation 8. Thus each PAA construction takes $O(\ell)$ time. In the following, we extend the running sum technique (cf. Section II-B) to construct PAA representations such that each construction takes only $O(\phi)$ time. We first expand Equation 8 (i.e., expanding $\hat{s}_i[x]$ by Equation 1) as follows.

$$e_{\hat{s}_i}[k] = \frac{\phi}{\ell} \sum_{x=\frac{\ell}{\phi}\cdot k}^{\frac{\ell}{\phi}(k+1)-1} \frac{s[i+x]-\mu(s_i)}{\sigma(s_i)} \tag{9}$$

where the value of $\mu(s_i)$ and $\sigma(s_i)$ can be computed in $O(1)$ time using two running sum arrays, A and A^2 (cf. Equations 5 and 6). To compute the summation form of $s[i+x]$ in $O(1)$ time, we derive the following equation based on the running sum array A.

$$\sum_{x=\frac{\ell}{\phi}\cdot k}^{\frac{\ell}{\phi}(k+1)-1} s[i+x] = A[i+\frac{\ell}{\phi}\cdot(k+1)-1] - A[i+\frac{\ell}{\phi}\cdot k] + s[i+\frac{\ell}{\phi}\cdot k] \tag{10}$$

Thus each ϕ-dimensional PAA representation can be constructed in $O(\phi)$ time. Algorithm 2 shows the pseudo-code for clarity.

Algorithm 2 Quick-Motif: Efficient PAA construction

```
Input: subsequence $s_i$, running sum arrays $A$, $A^2$, dimensionality $\phi$
Output: PAA representation $e_{\hat{s}_i}$
1: compute $\mu(s_i)$ and $\sigma(s_i)$ by $A$, $A^2$                      ▷ $O(1)$
2: for $k \leftarrow 0$ to $\phi-1$ do
3:     compute $\sum_{x=\frac{\ell}{\phi}\cdot k}^{\frac{\ell}{\phi}(k+1)-1} s[i+x]$ by $A$ using Equation 10   ▷ $O(1)$
4:     compute $e_{\hat{s}_i}[k]$ using Equation 9                          ▷ $O(1)$
```

In subsequent sections, we will utilize these PAA representations to support pruning. By [17], the PAA distance

d_{PAA} between two PAA representations $e_{\hat{s}_i}$ and $e_{\hat{s}_j}$ serves as the lower bound of the Euclidean distance between their representative subsequences $\hat{s}_i$ and $\hat{s}_j$:

$$d(\hat{s}_i, \hat{s}_j) \geq d_{PAA}(e_{\hat{s}_i}, e_{\hat{s}_j}) = \left(\frac{\ell}{\phi}\sum_{x=0}^{\phi-1}(e_{\hat{s}_i}[x] - e_{\hat{s}_j}[x])^2\right)^{\frac{1}{2}} \quad (11)$$

2) PAA MBR Construction: To support batch pruning, we attempt to group the PAA representations of subsequences into minimum bounding rectangles (MBRs) M. We only refine an MBR pair (i.e., calculating the distance of subsequence pairs in between two MBRs) if their minimum distance is smaller than the best-so-far distance *bsf*. The minimum distance between M_u and M_v is define as follows:

$$d_{PAA}(M_u, M_v) = \sqrt{\ell/\phi} \cdot d_2^{min}(M_u, M_v) \quad (12)$$

where $d_2^{min}(M_u, M_v)$ denotes the minimum 2-norm distance between two MBRs. Typically the grouping strategy is to group elements as tight as possible such that more unpromising MBR pairs can be pruned. However, if we group the subsequences arbitrarily (regardless of their offsets), then the running sum technique (c.f. Section II-B) cannot be applied and every pair-wise distance must be calculated from scratch. Thus the time complexity to process a surviving MBR pair is $O(|M_u||M_v|\ell)$, where $|M_i|$ indicates the number of elements in MBR M_i.

According to [11], [13], overlapping (i.e., trivially matched) subsequences tend to be similar. Thereby, grouping the PAA representation of overlapping subsequences likely minimizes the volume of MBRs. Another advantage of this grouping strategy is that the distance calculation in between two surviving MBRs can be reduced to $O(1)$ time by exploiting the locality of the elements. In the following, we formally define our grouping strategy and then introduce an efficient approach to calculate the subsequence distances by an incremental strategy.

w-MBR construction. We first define our grouping strategy, denoted as w-MBR, as follows.

Definition 2 (w-MBR, M_u^w): Given a sequence s of length m and motif length ℓ, a w-MBR M_u^w consists of all PAA representations $e_{\hat{s}_\tau}$ for all τ subject to $uw \leq \tau \leq (u+1)w-1$ and $\tau < m - \ell + 1$.

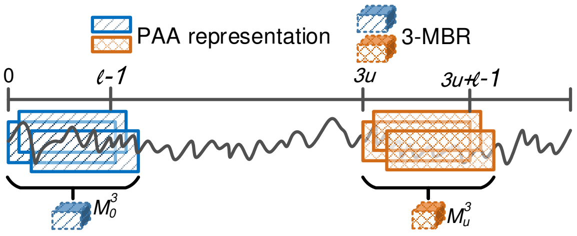


Fig. 4: An example of 3-MBRs construction

Figure 4 illustrates a concrete example of our grouping strategy, where we group every three consecutive PAA representations into a 3-MBR M_i^3 according to Definition 2. Thus the total number of MBRs is $(m - \ell + 1)/w$, and the space overhead is $O((m-\ell)\phi/w)$. To reduce the memory consumption in the implementation, we discard the PAA representation $e_{\hat{s}}$ after the corresponding w-MBR is constructed.

3) Refining a pair of w-MBRs: The w-MBR grouping strategy not only offers reasonable grouping performance (in terms of tightness) but also enables fast distance calculation of subsequence pairs. By applying the running sum techniques (cf. Section II-B), the pairwise distances of two w-MBRs can be calculated in $O(1)$ time incrementally.

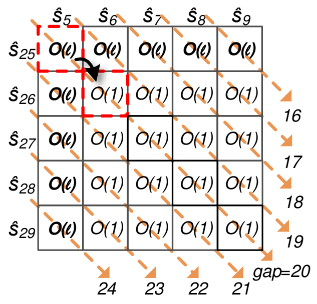

$\hat{s}_5$ $\hat{s}_6$ $\hat{s}_7$ $\hat{s}_8$ $\hat{s}_9$
$\hat{s}_{25}$ $\hat{s}_{26}$ $\hat{s}_{27}$ $\hat{s}_{28}$ $\hat{s}_{29}$
O(ℓ)
O(1)
16
17
18
19
gap=20
24
23
22
21

Fig. 5: Fast distance calculations of M_1^5 and M_5^5, $w = 5$

Figure 5 shows an example of the fast distance calculation in between M_1^5 and M_5^5. According to the discussion in Section II, $d(\hat{s}_6, \hat{s}_{26})$ can be derived from $d(\hat{s}_5, \hat{s}_{25})$ in $O(1)$ time using two running sum arrays. Thus, only 9 subsequence distances are calculated in its entirety (i.e., taking $O(\ell)$ time) and the other 16 subsequence distances can be computed in $O(1)$ incrementally as illustrated in Figure 5, where each incremental process is highlighted by one dashed line (i.e., diagonal cells). We conclude the effectiveness of this approach in Lemma 1.

Lemma 1: Given two w-MBRs, the ratio of full distance calculations is $\frac{2w-1}{w^2}$.

Proof: Trivial, the number of gap offsets in two w-MBRs is $2w - 1$ (i.e., the topmost row plus the leftmost column in Figure 5) and the total number of subsequence pairs is w^2. ■

Algorithm 3 Quick-Motif: Refining w-MBR pair

```
    Input: sequence $s$ of length $m$, motif length $\ell$, running sum arrays $A$,
    $A^2$; $w$-MBRs $M_u^w$, $M_v^w$.
    Output: motif offset $os$ and motif distance $bsf$
 1: $bsf \leftarrow \infty$
 2: for $gap \leftarrow (v-u-1)w+1$ to $(v-u+1)w-1$ do                ▷ $v \geq u$
 3:     $(s_i, s_j)$ is the first corresponding pair based on $gap$
 4:     if $s_i$ and $s_j$ is trivially matched then continue
 5:     repeat                                  ▷ process pairs with the same offset gap
 6:         compute $\mu(s_i)$,$\mu(s_j)$,$\sigma(s_i)$,$\sigma(s_j)$ by $A$, $A^2$              ▷ $O(1)$
 7:         if $(s_i, s_j)$ is the first subsequence pair then
 8:             compute $\sum_{x=0}^{\ell-1} s_i[x]s_j[x]$ in its entirety              ▷ $O(\ell)$
 9:         else
10:             compute $\sum_{x=0}^{\ell-1} s_i[x]s_j[x]$ incrementally by Eq. 7      ▷ $O(1)$
11:         compute $\rho(s_i, s_j)$ by Eq. 3                                      ▷ $O(1)$
12:         if $\sqrt{2\ell(1-\rho(s_i,s_j))} < bsf$ then
13:             $bsf \leftarrow \sqrt{2\ell(1-\rho(s_i,s_j))}$; $os \leftarrow (i,j)$
14:         $i \leftarrow i+1$; $j \leftarrow j+1$
15:     until $s_i \notin M_u^w$ or $s_j \notin M_v^w$
```

Inspired by the fast distance calculations, we study our first refinement algorithm (Algorithm 3) to discover a motif in between two w-MBRs. Note that we do not need to check the intra subsequence pairs inside a w-MBR since they are trivially-matched[4] when $w \leq \ell$. For each offset gap, we calculate the normalized distance of the first pair in its entirety and incrementally derive the distance of the remaining pairs with the same offset. In our running example of Figure 5, the first offset gap is 16 and it is easy to derive the first subsequence pair is $(\hat{s}_9, \hat{s}_{25})$. Note that this execution paradigm is memory-effective since it only requires keeping one cross sum (i.e., $\sum_{x=0}^{\ell-1} s_i[x]s_j[x]$) during the entire refinement process. According to Lemma 1, the time complexity of Algorithm 3 is $O(\frac{|M_u||M_v|\ell}{w}) = O(\frac{w^2\ell}{w}) = O(w\ell)$.

Discussion. The w-MBR grouping strategy not only offers effective pruning but also enables fast distance calculations. Obviously the parameter w plays a role in tuning the performance of Algorithm 3. According to Lemma 1, the ratio of complete distance calculations is lower when w is larger. However, the tightness of w-MBRs is also sensitive to the value of w which affects the pruning performance (to be discussed in the next section). We will experimentally demonstrate this tradeoff in Section IV.

B. *Filter-and-Refinement (FaR)*

In the last section we know how to discover a motif in a w-MBR pair. This section we turn our focus on how to efficiently find the surviving w-MBR pairs (i.e., their minimum distance is smaller than *bsf*). A naïve solution is to check every pair of w-MBRs where this solution takes $O(((m-\ell)/w)^2 \cdot \phi)$ time to compute. The motif discovery problem can be viewed as a problem of finding closest pair of ℓ-dimensional points. A better solution is to organize w-MBRs into a hierarchical structure and applies filter-and-refinement framework to discover the surviving w-MBR pairs.

Hilbert R-tree, H^ξ. In this work, we simply apply Hilbert R-tree to group w-MBRs since it offers reasonable grouping quality and fast construction time [18]. In motif discovery, the construction cost remains a performance factor since the query sequences may arrive on-the-fly.

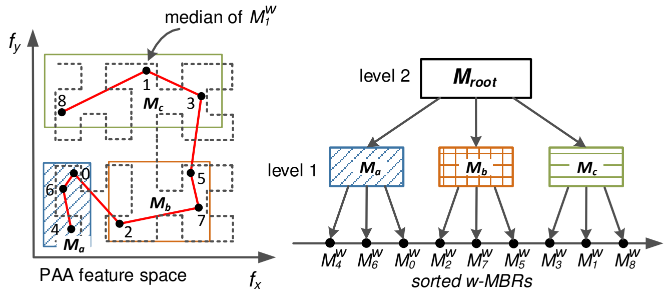
We place the Hilbert R-tree in memory. Instead of using the page size, let ξ be the branch factor (system parameter). We first sort the w-MBRs based on their Hilbert curve order. Based on the sorted order, we group every ξ MBRs into a next-level MBRs. The grouping process is recursively called until there is only one MBR constructed at i-th level. As an example illustrated in Figure 6(a), there are 9 w-MBRs and ξ is set to 3. We construct 3 level Hilbert R-tree. The space overhead is negligible as the number of non-leaf nodes is much smaller than the number of leaf nodes (i.e., w-MBRs).

State-of-the-art. Sorted-list aggregation [21] and best-first search [19], [20] are the state-of-the-art solutions in finding

TABLE II: Statistics of methods on TAO dataset under default setting

	Locality-based	Best-first [19], [20]	Sorted-list [21]
Non-leaf pairs	18.11 M	18.14 M	N/A
Surviving pairs	0.1256 M	0.1249 M	0.1249 M
Heap size	N/A	2.78 M	0.075 M
# pushes	11.73 M (queue)	6.75 M (heap)	2357 M (heap)
Resp. time	1.56 s	6.32 s	1409.1 s

[4]In our framework, we always set w smaller than ℓ.



(a) Example of a Hilbert R-tree, H^3, $\xi = 3$

Level 1	$M_a \rightarrow (M_4^w, M_6^w), (M_4^w, M_0^w), (M_6^w, M_0^w)$ $M_b \rightarrow (M_2^w, M_7^w), (M_2^w, M_5^w), (M_7^w, M_5^w)$ $M_c \rightarrow (M_3^w, M_1^w), (M_3^w, M_8^w), (M_1^w, M_8^w)$
Level 2	$M_{root} \rightarrow (M_a, M_b), (M_a, M_c), (M_b, M_c)$ $(M_a, M_b) \rightarrow (M_4^w, M_2^w), (M_4^w, M_7^w), (M_4^w, M_5^w) \cdots$ $(M_a, M_c) \rightarrow (M_4^w, M_3^w), (M_4^w, M_1^w), (M_4^w, M_8^w) \cdots$ $(M_b, M_c) \rightarrow (M_2^w, M_3^w), (M_1^w, M_7^w), (M_2^w, M_8^w) \cdots$

(b) Search space in each level

Fig. 6: Hilbert R-tree and its search space

closest pair. Both approaches utilize min-heaps to prioritize the execution order such that the number of surviving w-MBR pairs can be minimized. However, the cost of the heap operations may already outweigh its benefit in motif discovery due to the problem dimensionality. For instance, [21] only evaluates their method up to 6 dimensions. For clarity, we demonstrate the performance of these two methods in Table II on TAO dataset. These two methods execute excessive push operations in the min-heap(s). To make things worse, the best-first search may introduce many false positive MBR pairs into the min-heap due to the curse of dimensionality (i.e., ϕ). The best-first search maintains a min-heap with 2.78 million elements on average. Alternatively, the sorted-list aggregation requires to maintain ϕ min-heaps (instead of one) and examines the candidate pairs one-by-one from ϕ min-heaps. The lack of batch pruning and the curse of dimensionality make the sorted-list aggregation execute more push operations than the best-first search.

This analysis raises an interesting question, *can we minimize surviving w-MBR pairs without costly heap operations?*

Locality-based search strategy. In this work, we recommend to execute the filtering phase using a locality-based strategy. The locality-based search strategy prioritizes the execution order by exploring the *locality* of the Hilbert R-tree, and it only utilizes a queue during the execution (i.e., taking $O(1)$ time for each operation). We begin by defining the $LLCA$ value of two w-MBRs.

Definition 3 ($LLCA(M_u^w, M_v^w)$): Given two w-MBRs, M_u^w and M_v^w, we define $LLCA(M_u^w, M_v^w)$ as the level of their lowest common ancestor.

For example, $LLCA(M_4^w, M_6^w) = 1$ and $LLCA(M_4^w, M_3^w) = 2$ in Figure 6(a). The basic idea of the locality-based search strategy is to examine w-MBR pairs in ascending order of their $LLCA$ values. In other words, we prioritize to examine w-MBR pairs from closest to furthest in terms of their tree path distances. As illustrated in Figure 6(b), the search space of all tree nodes at level i covers the all w-MBR pairs with their $LLCA$ value equal to i (cf. line 3 in Algorithm 4); When processing M_{root} at level 2, we generate 3 intermediate pairs, $(M_a, M_b), (M_a, M_c), (M_b, M_c)$. Suppose (M_a, M_b) is the only surviving pair whose distance lower bound is smaller than *bsf*, we then examine 9 inter w-MBR pairs in between M_a and M_b, e.g., (M_4^w, M_2^w). The pseudo code of the locality-based search strategy is shown in Algorithm 4 for clarity. Algorithm 4 covers all possible w-MBR pairs which provides the exactness of our search strategy.

Algorithm 4 Quick-Motif: Filter-and-refinement

```
Input: sequence $s$ of length $m$, motif length $\ell$, pre-computed cumulative
arrays $A$ and $A^2$, Hilbert R-tree $H^\xi$
Output: motif offset $os$ and motif distance $bsf$
1:  $bsf \leftarrow \infty$
2:  for level $i$ from 1 to root do
3:    for $M_j$ at level $i$ do        ▷ find all $w$-MBR pairs with $LLCA = i$
4:      for all children pairs $(M_u, M_v)$ of $M_j$ do
5:        push $(M_u, M_v)$ into queue $Q$ if $d_{PAA}(M_u, M_v) < bsf$
6:      while $Q$ is not empty do
7:        pop a pair $(M_u, M_v)$ from Q
8:        if $M_u, M_v$ are not the leaf nodes then
9:          for all inter pairs $(M'_u, M'_v)$ of $M_u$ and $M_v$ do
10:           push $(M'_u, M'_v)$ into $Q$ if $d_{PAA}(M'_u, M'_v) < bsf$
11:       else
12:         invoke Algorithm 3 to refine $(M_u, M_v)$
13:         update $os$ and $bsf$ if a better motif is found
```

As shown in Table II, the locality-based approach introduces slightly more surviving w-MBR pairs, but it completely liberates from the costly heap operations. When compared with the best-first search, the locality-based approach examines fewer non-leaf node pairs as the running *bsf* decreases faster due to the locality of the result. As a consequence, the locality-based approach outperforms other two search strategies.

C. Lazy Group Refinement (LGR)

1) Limitation of FaR: The filter-and-refinement framework (Algorithm 4) is an efficient solution especially when it prunes large amount unpromising MBR pairs at the filtering phase. However, the pruning ratio is sensitive to the data distributions. In our experimental testings, the pruning ratio can be from 75.24% in EPG to 99.99% in TAO. In the worst case, it has $\Delta(\Delta-1)/2$ w-MBR pairs to invoke Algorithm 3, where Δ indicates the total number of w-MBRs (i.e., $(m-\ell)/w$). If there are many surviving w-MBR pairs (e.g., 20%) from the filtering phase, then the refinement phase becomes costly as refining a w-MBR pair takes $O(w\ell)$ time.

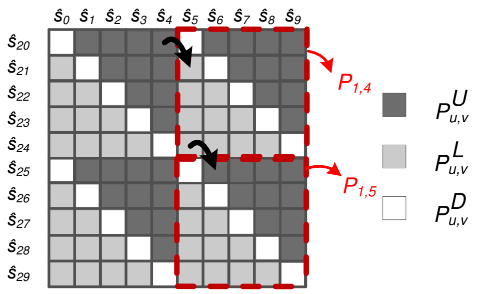

$\hat{s}_0$ $\hat{s}_1$ $\hat{s}_2$ $\hat{s}_3$ $\hat{s}_4$ $\hat{s}_5$ $\hat{s}_6$ $\hat{s}_7$ $\hat{s}_8$ $\hat{s}_9$
$\hat{s}_{20}$ $\hat{s}_{21}$ $\hat{s}_{22}$ $\hat{s}_{23}$ $\hat{s}_{24}$ $\hat{s}_{25}$ $\hat{s}_{26}$ $\hat{s}_{27}$ $\hat{s}_{28}$ $\hat{s}_{29}$
$P_{1,4}$
$P_{1,5}$
$P^U_{u,v}$
$P^L_{u,v}$
$P^D_{u,v}$

Fig. 7: Lazy Group Refinement

2) Concept of Lazy Group Refinement: Inspired by the above discussion, we attempt to refine the surviving w-MBR pairs by batch such that the reusability of running cross-sum is maximized (cf. Line 10 of Algorithm 3). Figure 7 illustrates this idea by an example. For ease of presentation, we use $P_{u,v}$ to indicate a w-MBR pair (M_u^w, M_v^w). Suppose there are 4 w-MBR pairs (e.g,. $P_{0,4}$, $P_{0,5}$, $P_{1,4}$, and $P_{1,5}$) to be refined by Algorithm 3. If we process every w-MBR pair individually, each pair has to calculate 9 out of 25 distances in their entirety (cf. Section III-A2). However, if we lazily refine these 4 pairs in a batch, then some distance calculations can incrementally computed in $O(1)$ time. For instance, $d(\hat{s}_5, \hat{s}_{21})$ can be derived from $d(\hat{s}_4, \hat{s}_{20})$ in $O(1)$ time if we refine $P_{0,4}$ and $P_{1,4}$ in one batch. For clarity, $d(\hat{s}_5, \hat{s}_{21})$ is necessarily calculated in its entirety if we process $P_{1,4}$ individually.

Sub-Partition of $P_{u,v}$. To support LGR, we first partition the calculation space of a w-MBR pair (i.e,. the rectangles in Figure 5 and Figure 7) into 3 sub-partitions. Without loss of generality, we assume that v is always larger than u since $P_{u,v}$ and $P_{v,u}$ are interchangeable. The first sub-partition (*upper partition*), $P^{\mathcal{U}}_{u,v}$, indicates all subsequence pairs with offset gap range from $(v-u-1)w+1$ to $(v-u)w-1$ (e.g., the upper triangle in Figure 7); the second sub-partition (*lower partition*), $P^{\mathcal{L}}_{u,v}$, indicate all subsequence pairs with offset gap range from $(v-u)w+1$ to $(v-u+1)w-1$ (e.g., the lower triangle in Figure 7); and the last sub-partition (*diagonal partition*), $P^{\mathcal{D}}_{u,v}$, contains all subsequence pairs with offset gap $(v-u)w$ (e.g. the diagonal elements in Figure 7). Given the sub-partition notations, we study the following Lemmas which secure the performance gain of the lazy refinement process.

Lemma 2: The distance of subsequence pairs in $P^X_{u,v}$ can be calculated in $O(1)$ time if
(i) $P^{\mathcal{L}}_{u,v-1}$ is processed together when $X=\mathcal{U}$; or
(ii) $P^{\mathcal{U}}_{u-1,v}$ is processed together when $X=\mathcal{L}$; or
(iii) $P^{\mathcal{D}}_{u-1,v-1}$ is processed together when $X=\mathcal{D}$;

Proof: Due to the space limit, we only show the correctness of the third condition. The other two conditions can be proved similarly. The subsequence pairs in $P^{\mathcal{D}}_{u,v}$ have only one offset gap $(v-u)w$. Thus the cross sum of the first subsequence pair $\{s_{uw}, s_{vw}\}$ in $P^{\mathcal{D}}_{u,v}$ can be incrementally calculated from the last subsequence pair $\{s_{uw-1}, s_{vw-1}\}$ located in $P^{\mathcal{D}}_{u-1,v-1}$ based on Equation 7. ∎

For instance, the distance of $P^{\mathcal{U}}_{1,5}$ (e.g., $d(\hat{s}_6, \hat{s}_{25})$) can be calculated in $O(1)$ time if it is processed with $P^{\mathcal{L}}_{1,4}$ (e.g., $d(\hat{s}_5, \hat{s}_{24})$). We formally give the performance gain of LGR in Lemma 3.

Lemma 3: Given n adjacent sub-partitions to be processed in a batch, the ratio of full distance calculations is $\frac{1}{nw}$ when the process starts from $P^{\mathcal{D}}_{u,v}$; otherwise, the ratio is $\frac{2}{nw}$.

Proof: For diagonal partition, each partition has w subsequence pairs to be processed and only one pair is computed in it entirety. Thus, the ratio of full distance calculations is $\frac{1}{nw}$ if we process n adjacent diagonal partitions by batch.

For other two types of partitions, each partition has $w(w-1)/2$ subsequence pairs and it requires to calculate $w-1$ subsequence pairs in their entirety (i.e., the first row or the first column pairs). Thus, the ratio of full distance calculations is $\frac{w-1}{nw(w-1)/2} = \frac{2}{nw}$. ■

For instance, when we lazily refine 4 w-MBR pairs (i.e., 4 upper, 4 lower, and 4 diagonal partitions) in Figure 7, the number of full distance calculations is reduced to $2*10-1 = 19$ [5] from $4*(2*5-1) = 36$. Thus the full distance calculations is 52.7% smaller than refining w-MBRs individually. More importantly, the performance gain provided by the lazy group refinement becomes more obvious when more w-MBR pairs are surviving from the filtering phase (such that more adjacent sub-partitions are processed by batch).

3) Filter-and-Refinement with LGR: We still need an effective strategy to integrate LGR into the filter-and-refinement framework. Note that we cannot simply defer every w-MBR pair (line 12 of Algorithm 4) to LGR as this naïve strategy never updates bsf in the filtering phase. An alternative strategy is to defer a w-MBR pair only when their lower bound is smaller than a given threshold. However, there is no unified approach to define the threshold for different datasets. To make our solution more data-oriented, we set the best-so-far value bsf after processed λ levels in FaR as bsf_λ. More specifically, tree nodes with level smaller than λ would be filtered and refined using Algorithm 4 (cf. Line 2), and the correlation of motif discovered among these tree nodes will be used as a fix threshold (i.e., bsf_λ) to derived the candidate w-MBR pairs for LGR. The value of λ is typically set to a small value (e.g., λ=2) as the locality of the Hilbert R-tree can provide a good enough threshold at low level.

Given all deferred w-MBR pairs (whose lower bounds are smaller than bsf_λ), their subsequence distances can be computed by batch using LGR. According to Lemma 2, the distance of a sub-partition can be calculated in $O(1)$ time if it is processed with its predecessor. For instance, we can compute the normalized distance of $P^{\mathcal{U}}_{1,5}$ in $O(1)$ time if it is processed with $P^{\mathcal{L}}_{1,4}$ in the same batch. Straightforwardly we can *map* all surviving w-MBR pairs into a 2D array (with size $\Delta \times \Delta$) and dig out the adjacent sub-partitions by scanning the 2D array from top-left to right-bottom. The space overhead of this solution is $O(\Delta^2)$. However, Δ^2 can be very large in practices (based on the length of input sequence and w). For instance, the value of Δ^2 is 1.0×10^{10} in the default setting of our experiments where the space overhead is 37.25 GB if every cell store a 4 Bytes record. In the following, we discuss a new solution which can reduce the space overhead to $O(\Delta + N_{survive})$, where $N_{survive}$ indicates the number of surviving pairs.

To efficiently dig out the preceding information, we maintain two array lists L_{first} and L_{last} where each list contains 2Δ buckets in total. Each bucket keeps the preceding information of its corresponding offset gap interval. For instance, the first two buckets keep the preceding information of interval $[1, w-1]$ and $[w, w]$, respectively (denoted as $0w^+$ and $1w$ in short). For clarity, we illustrate the relationship of the buckets and the offset gaps in Figure 8.

Figure 9 is a concrete example of 5 surviving w-MBR pairs to demonstrate how to process all adjacent sub-partitions by

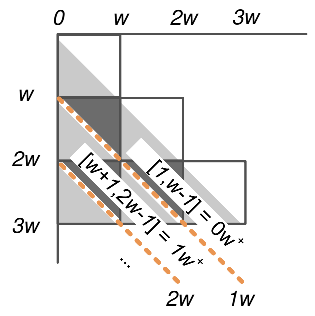
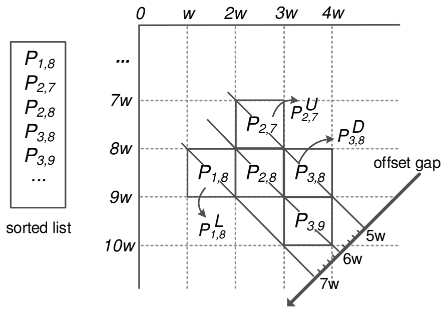


Fig. 8: The relationship of buckets and offset gaps



(a) Sorted list and corresponding search space

	...	5w$^+$	6w	6w$^+$	7w	7w$^+$	...
L_{first}	...	-	-	$P^{\mathcal{U}}_{1,8}$	$P^{\mathcal{D}}_{1,8}$	$P^{\mathcal{L}}_{1,8}$	...
L_{last}	...	-	-	$P^{\mathcal{U}}_{1,8}$	$P^{\mathcal{D}}_{1,8}$	$P^{\mathcal{L}}_{1,8}$	...

(b) 1st access, $P_{1,8}$

	...	5w$^+$	6w	6w$^+$	7w	7w$^+$	...
L_{first}	...	$P^{\mathcal{L}}_{2,7}$	$P^{\mathcal{D}}_{2,8}$	$P^{\mathcal{U}}_{1,8}$	$P^{\mathcal{D}}_{1,8}$	$P^{\mathcal{L}}_{1,8}$	...
L_{last}	...	$P^{\mathcal{U}}_{2,8}$	$P^{\mathcal{D}}_{2,8}$	$P^{\mathcal{L}}_{2,8}$	$P^{\mathcal{D}}_{1,8}$	$P^{\mathcal{L}}_{1,8}$	...

(c) 3rd access, $P_{2,8}$

	...	5w$^+$	6w	6w$^+$	7w	7w$^+$	...
L_{first}	...	$P^{\mathcal{L}}_{2,7}$	$P^{\mathcal{D}}_{2,8}$	$P^{\mathcal{L}}_{3,9}$	$P^{\mathcal{D}}_{1,8}$	$P^{\mathcal{L}}_{1,8}$	...
L_{last}	...	$P^{\mathcal{U}}_{3,9}$	$P^{\mathcal{D}}_{3,9}$	$P^{\mathcal{L}}_{3,9}$	$P^{\mathcal{D}}_{1,8}$	$P^{\mathcal{L}}_{1,8}$	...

(d) Refine sub-partitions $P^{\mathcal{U}}_{1,8}$ to $P^{\mathcal{L}}_{2,8}$ after 5th access, $P_{3,9}$

Fig. 9: An example of lazy group refinement

batch using the preceding arrays. During the filtering phase, we insert every deferred w-MBR pair into a sorted list where the pairs are sorted in an order to their (u, v) values. For instance, $P_{1,8}$ is prioritized ahead of $P_{1,9}$ and $P_{2,7}$ in the sorted list. After we defer all surviving w-MBR pairs to LGR, we iteratively process these pairs according to the sorted list. For each processing pair $P_{u,v}$, we decompose it into 3 sub-partitions, $P^{\mathcal{U}}_{u,v}$, $P^{\mathcal{D}}_{u,v}$, and $P^{\mathcal{L}}_{u,v}$, and then maintain the corresponding bucket of the preceding arrays. For instance, when getting $P_{1,8}$ at the first access, it is decomposed into $P^{\mathcal{U}}_{1,8}$, $P^{\mathcal{D}}_{1,8}$, and $P^{\mathcal{L}}_{1,8}$ where their corresponding buckets are $6w^+$, $7w$, and $7w^+$, respectively (cf. Figure 9(a)). As an example, we simply mark $P^{\mathcal{U}}_{1,8}$ as a new batch in bucket $6w^+$ (by updating L_{start} and L_{end} in Figure 9(b)) as bucket $6w^+$ is *empty*. In the next iteration, we get pair $P_{2,7}$ from the sorted list and update the preceding information of *empty* buckets $4w^+$, $5w$, and $5w^+$. When processing the next pair $P_{2,8}$, we update the

[5]For ease of understanding, this number can be derived from the first row plus the first column if the refining w-MBRs is grouped as a big rectangle.

preceding information of $5w^+$, $6w$, and $6w^+$. Note that we update L_{last} of $5w^+$ since $P^{\mathcal{U}}_{2,8}$ is adjacent to $P^{\mathcal{L}}_{2,7}$ in $5w^+$ and the reason of updating L_{last} of $6w^+$ is similar.

The preceding information are iteratively updated till the new coming sub-partition is not adjacent to the previous batch. For instance, when getting $P^{\mathcal{L}}_{3,9}$, the preceding information in $6w^+$ (from $P^{\mathcal{U}}_{1,8}$ to $P^{\mathcal{L}}_{2,8}$) is not the predecessor of $P^{\mathcal{L}}_{3,9}$. We refine $P^{\mathcal{U}}_{1,8}$ to $P^{\mathcal{L}}_{2,8}$ by batch and replace the preceding information by $P^{\mathcal{L}}_{3,9}$ (cf. Figure 9(d)). Note that we do not miss any adjacent subpartition in the batch processing since all pairs are sorted access from top-left to right-bottom (as illustrated in Figure 9(a)).

Algorithm 5 Quick-Motif: Lazy group refinement

```
Input: sequence $s$ of length $m$, motif length $\ell$, pre-computed cumulative arrays $A$ and $A^2$, sorted list $SL$
Output: motif offset $os$ and motif distance $bsf$
1: initial $L_{begin}, L_{end}$ of $2\Delta$ buckets to $null$
2: for $P_{u,v} \in SL$ do
3:     for each $P^X_{(u,v)}$, $X \in \mathcal{U}, \mathcal{L}, \mathcal{D}$ do
4:         $\mathcal{B}[idx] \leftarrow$ bucket of $P^X_{(u,v)}$                ▷ Equation 13
5:         $(L_{begin}, L_{end}) \leftarrow \mathcal{B}[idx]$'s preceding information
6:         if $L_{end}$ is the predecessor of $P^X_{(u,v)}$ then             ▷ Lemma 2
7:             $L_{end} \leftarrow P^X_{(u,v)}$
8:         else
9:             process sub-partitions from $L_{begin}$ to $L_{end}$ in $\mathcal{B}[idx]$
10:            $L_{begin} \leftarrow P^X_{(u,v)}$; $L_{end} \leftarrow P^X_{(u,v)}$
```

Algorithm 5 shows the pseudo code for LGR. The sorted list SL keeps all surviving w-MBR pairs which is maintained by a revised Algorithm 4. Given a subpartition $P^X_{u,v}$, the index of the corresponding bucket (line 4 of Algorithm 5), idx, is easily derived from a simple rule using u, v, and X as follows.

$$idx = \begin{cases} (v-u-1)w^+ & \text{if X=}\mathcal{U} \\ (v-u)w & \text{if X=}\mathcal{D} \\ (v-u)w^+ & \text{if X=}\mathcal{L} \end{cases} \quad (13)$$

In Line 9, processing adjacent sub-partitions in batch is just a variant of refining an w-MBR pair (Algorithm 3). The detail is omitted for simplicity.

To use LGR, we need to slightly modify the filter and refinement process (lines 12-13) of Algorithm 4. If the process level i is smaller than the level constraint (i.e., λ), then we invoke Algorithm 3 and update bsf value. Otherwise, we insert the surviving w-MBR pair into a sorted list SL. After the filter and refinement process, Algorithm 5 is invoked using SL as input.

Discussion. As compared with the 2D array solution, our approach consumes much smaller space in practices since we only keep two array lists plus one sorted list instead of a 2D array, $O(\Delta^2)$ vs $O(\Delta + N_{survive})$. Even though $N_{survive}$ can be up to $\Delta(\Delta-1)/2$, we can avoid the worst case by fine-tuning the level constraint (i.e., λ) during the execution.

In majority of cases, applying LGR into the filter-and-refinement framework (FaR) is beneficial since the only overhead is to prepare the sorted list of the surviving w-MBR pairs. Obviously, the sorting cost becomes higher when there are more deferred w-MBR pairs. However, such large amount of deferred w-MBR pairs will improve the refinement performance by batch processing (Line 9 of Algorithm 5 and Lemma 3). Thereby, we strongly recommend to apply LGR into FaR as a complete framework for Quick-Motif.

D. Putting it all together

Algorithm 6 Quick-Motif

```
Input: A sequence $s$ of length $m$, motif length $\ell$, dimensionality $\phi$, grouping size $w$, branch factor $\xi$, level constraint $\lambda$
Output: motif offset $os$ and motif distance $bsf$
1: Precompute running sum array $A$, $A^2$                          ▷ Section II-B
2: Construct $w$-MBRs , ($w$ and $\phi$)                            ▷ Section III-A
3: Construct Hilbert R-tree, ($\xi$)                                ▷ Section III-B
4: Invoke Alg. 4 to filter and refine $w$-MBR pairs, ($\lambda$)     ▷ Section III-B
5: Invoke Alg. 5 to process $w$-MBR pairs by batch                  ▷ Section III-C
```

We are now ready to present our complete framework, Quick-Motif, in Algorithm 6. We first construct two running sum arrays using the techniques introduced in Section II-B. Next we construct the w-MBRs of the query sequence (cf. Section III-A). To filter out unpromising w-MBR pairs, we organize the w-MBRs into a Hilbert R-tree and then apply the locality-based search strategy (cf. Section III-B). Lastly we apply our lazy group refinement technique to process the surviving w-MBR pairs by batch (cf. Section III-C).

Parallel execution. Quick-Motif is a parallel friendly framework. In the filter-and-refinement process, the master dispatchs each processing node (cf. line 3 of Algorithm 4) to different slaves. When a slave returns a result, the master maintains the sorted list SL and motif result if necessary. In LGR, the master partitions the SL into several pieces according to the offset gap and dispatch them to different salves so that each slave invokes their LGR locally. Note that this parallel framework unlikely reduces the effectiveness of the batch processing since w-MBR pairs of each small piece remain sorted.

IV. Experiments

In this section, we conduct extensive experiments to evaluate Quick-Motif (QM) with other competitors. All methods were implemented in C++, complied using gcc 4.6.3, and performed on a machine equipped with Intel Core 6-Cores (12-Threads) i7-3930K 3.2GHz and 16GB main memory. The machine was running Ubuntu 12.04. For the sake of experimental reproducibility, we have posted our datasets and executables at [22].

A. Experiment Setup

We use both synthetic and real datasets in the experiments. The synthetic datasets are used to evaluate the scalability of our techniques. Each sequence is generated by a random model (adopted in [5], [17], [23]) as follows.

$$s[i+1] = s[i] + N(\mu, \sigma) \quad (14)$$

where $N(\mu, \sigma)$ is a normal distribution function. By default, we set the mean μ=0 and the standard deviation σ=1 (following the experiment setting of [5]). Besides, we evaluate our work on four real datasets that were used in [3], [5], [10], [11].

ECG	The Koski ECG was evaluated in [10] where the ECG sequence is of length $144,002$.

EEG	It is a sequence of length 180,204 which can reflect the activity of large populations of neurons [5].
EPG	It is a concatenation of the two EPG sequences which traces insect behavior [3], it has 106,950 observations.
TAO	It records the sea surface temperatures of 55 array sensors over years. We evaluate our methods on a sequence of length 374,071 collected from the array sensors. This dataset was used in [11].

TABLE III: Experimental parameters and their values

Parameter	Default	Range
Sequence length, m	500k	250k, 500k, 750k, 1000k
Motif length, ℓ	500	300, 500, 700, 900
Grouping size, w/ℓ	1%	0.5%, 1%, 2%, 3%, 4%
Branch factor, ξ	4	2, 4, 6, 8
Level constraint, λ	2	1, 2, 3, 4

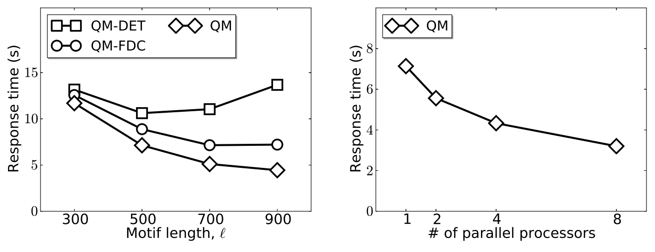
Table III summarizes the ranges of the investigated parameters and their default values. In each experiment, we vary a single parameter, while setting the others to their default values. For experiments on synthetic dataset, we report the average response time by running the methods on 10 sequences. We compare QM to two closest competitors MK [5] and SBF [9]. For MK, 20 subsequences are randomly selected to build the indices as investigated by authors in [5]. In QM, we set the PAA dimensionality ϕ to 10 as suggested in [11] and the tuning of other three system parameters w/ℓ, ξ and λ will be studied at Section IV-D. The response time of MK and QM include the indices construction time, and for each testing, we only use one core to run algorithms unless explicitly stated.

As a remark, under the default settings, the construction time of MK is 7.17 s (0.18% of overall cost) and it is only 0.39 s (5.5% of overall cost) for QM. The memory usage of SBF, MK, and QM is 13 MB, 2273 MB and 48 MB, respectively. QM uses 35 MB (out of 48 MB) extra memory to support pruning and lazy group refinement (cf. Sec. III-C).

B. Effectiveness of Optimizations

We have proposed an optimization for filtering w-MBR pairs, and evaluated it in Table II (in Section III-B). In this subsection, we proceed to examine the effectiveness of two optimization techniques for refining subsequence pairs: (i) fast distance calculation (Alg. 3 and noted as QM-FDC) and (ii) lazy group refinement (Alg. 6). These two techniques are compared with *distance early termination* (noted as QM-DET). Instead of invoking Alg. 3 to refine a surviving w-MBR pair as in QM-FDC, QM-DET uses the distance early termination to refine subsequence pairs in w-MBR pairs (cf. line 12 of Alg. 4). In order to utilize the distance early termination, we need to keep all normalized subsequences in the memory [4], and this makes QM-DET infeasible for very long sequences.

Effectiveness of FDC. Even distance early termination can be quite effective when tight threshold is applied, it still needs building time and memory space for all normalized subsequences. As shown in Figure 10(a), the overall performance of QM-FDC is superior to QM-DET , and the superiority of



(a) Effect of motif length, ℓ (b) Motif in parallel

Fig. 10: The effect of optimization methods, on synthetic dataset

QM-FDC becomes significant when ℓ increases, e.g., 1.9 times faster when $\ell = 900$.

Effectiveness of LGR. The performance gap between QM-FDC and QM is shown in Figure 10(a). The tight correlation of motif, e.g., 0.993 when $\ell = 500$, led to fewer sub-partitions for LGR, thus the performance gain of QM is limited. However, QM is 1.62 times faster than QM-FDC when $\ell = 900$ as it has more adjacent sub-partitions for LGR with the decreasing of motif's correlation in this setting.

Parallel execution. Next we evaluate the parallel version of QM in Figure 10(b), where it is implemented in OpenMP[6]. Experimental result demonstrates that the response time decreases when more cores are used. For example, the response time is reduced to 3.2 s (from 7.13 s) when we execute QM by 8 cores (compared to single core).

C. Performance Studies

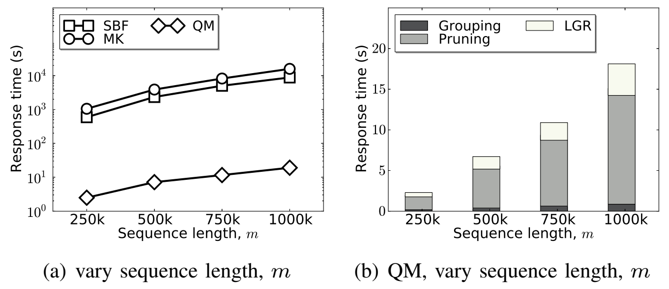
Overview. In this subsection, we compare QM with two state-of-the-art solutions, MK and SBF. QM outperforms these two methods (i.e., up to 3 orders of magnitude faster) in all performance studies. The improvement of QM makes online processing feasible for motif discovery (e.g., 2.48 s at sequence length m=250k).

Scalability experiments on synthetic data. Figure 11(a) shows the response time of the methods as a function of query sequence length m, after setting all other parameters to their default values. Cost grows with m for all methods. The result of QM is impressive since it is at least two orders of magnitude faster than SBF and MK. For instance, QM is 327 and 543 times faster than SBF and MK, respectively, when m=500k.

Figure 11(c) shows the response time of the methods versus motif length ℓ. SBF is not very sensitive to ℓ since SBF incrementally computes the distance of $(m-\ell)^2$ pairs (i.e., taking $O((m-\ell)^2)$ time). MK is also not very sensitive to ℓ since the length of subsequences affects the pruning ability of the reference indices. QM is the only method that has obvious gain when ℓ becomes larger since it jointly reduces the number of comparisons and accelerates each individual correlation calculation. Similarly, QM is at least two orders of magnitude faster than SBF and MK.

Figure 11(b), 11(d) illustrate the cost of each phase in QM. *Grouping* is the cost for constructing Hilbert R-tree;

[6]http://openmp.org/wp/


(a) vary sequence length, m (b) QM, vary sequence length, m

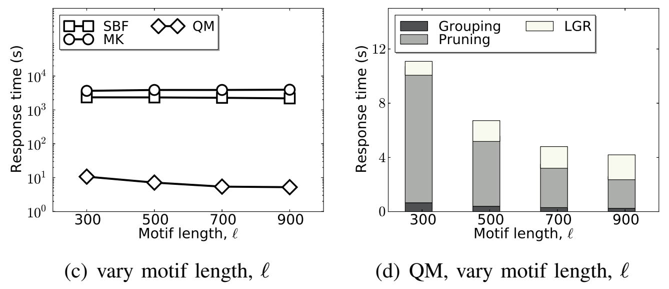
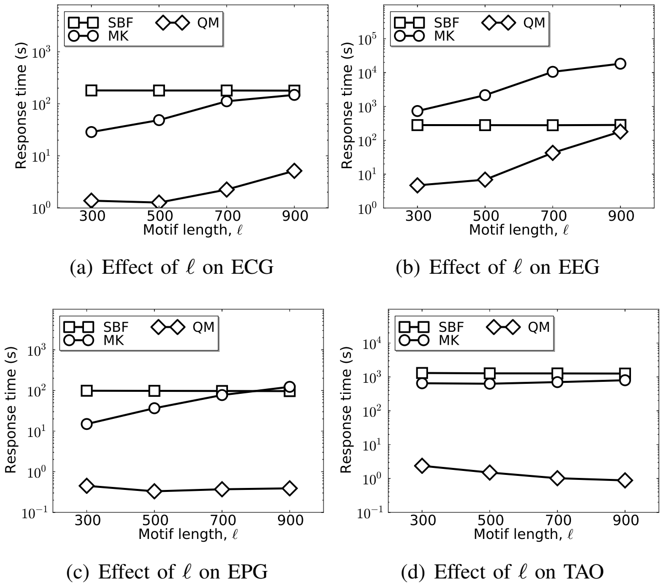

(c) vary motif length, ℓ (d) QM, vary motif length, ℓ

Fig. 11: Scalability evaluation, on synthetic dataset

Pruning represents the cost to prune unpromising w-MBR pairs and prepare the sorted list; *LGR* is the cost for our lazy group refinement. *Grouping*, *Pruning* and *LGR* occupies 5.5%, 72.8%, 21.7% of cost time in the default setting, respectively. As shown in Figure 11(b), by varying m, the cost distributions of these three phases are similar as using the same ℓ. When ℓ becomes larger, QM prefers to use larger w (e.g., $w = 0.01\ell$), thus fewer w-MBR pairs under the same m setting and the cost of *Pruning* reduces as in Figure 11(d).

Real data experiments. Figure 12 plots the response time


(a) Effect of ℓ on ECG (b) Effect of ℓ on EEG

(c) Effect of ℓ on EPG (d) Effect of ℓ on TAO

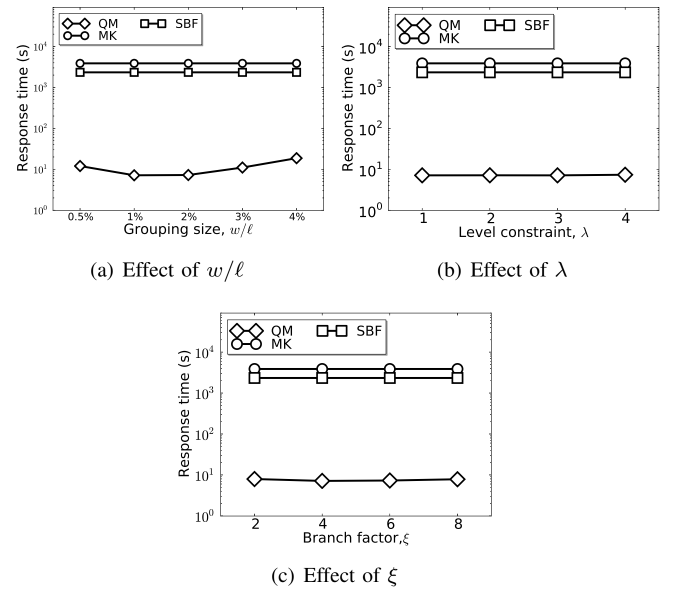
Fig. 12: Experiments on real datasets

of all methods versus motif length ℓ on four real datasets. In TAO and ECG, MK is superior to SBF for all motif length since the reference indices can provide competitive pruning ability. Again QM outperforms MK and SBF, e.g., in TAO dataset, QM is 1412 and 897 times faster than SBF and MK, respectively, when ℓ = 900. As a remark, in EEG dataset, QM-FDC performs slower than SBF as the pruning ability is weak when $\ell = 900$. Our best method QM is 2.93 times faster than QM-FDC due to the gain of LGR, and still performs better than SBF as in shown in Figure 12(b).

D. Tuning of System Parameters

In this subsection, we test the robustness of QM by varying three system parameters: w/ℓ, λ and ξ. According to our experimental evaluation, QM is not very sensitive to λ and ξ, and the only necessary tuning parameter is w. However even using the worst w/ℓ setting, QM is at least two orders of magnitude faster than SBF and MK.

Effect of w/ℓ. We first study the effect of the grouping size


(a) Effect of w/ℓ (b) Effect of λ

(c) Effect of ξ

Fig. 13: Tuning of system parameters, on synthetic dataset

w, where w controls the number of consecutive subsequences grouped in a MBR. To make our system suitable for different length motif discovery, this parameter is set to a ratio of ℓ. Figure 13(a) shows the effect of w. Obviously the value of w leads a tradeoff between the effectiveness of pruning capability and fast distance calculations. When w is set to a small value (e.g., $w/\ell = 0.5\%$), the filter-and-refinement can prune more MBR pair due to the tightness of the MBRs but it degrades the effectiveness of the fast distance calculations. We should use a small value of w; otherwise it may introduce many w-MBR pairs to be refined (such that more subsequence pairs to be computed). According to our experiments, we set $w/\ell = 1\%$ in our default setting as it achieves good overall performance.

Effect of λ. Next we demonstrate the effect of the level constraint λ in LGR in Fig. 13(b). As explained in Section III-C, the value of λ can be set to a small value (e.g., 2). More importantly, the response time is not very sensitive to λ which verifies the robustness of our grouping idea and the bottom-up search strategy (i.e., a good motif candidate can be discovered in low levels of w-MBR pairs).

Effect of ξ. Fig. 13(c) shows the effect of branch factor ξ. According to the experimental evaluation, the performance of QM is not very sensitive to ξ, and we set $\xi = 4$ in our default setting.

V. Other Related Work

Time series motif discovery was first introduced by Chiu et at [13]. Now it becomes a core subroutine in modern sequence mining tasks [3], [7]. As introduced in Section I, the naïve solution requires quadratic number of distance calculations which becomes infeasible for large scale data in modern applications.

To address the performance of motif discovery, one direction is to discover an approximate motif. Castro and Azevedo [1] propose an approximate solution that converts a sequence into a set of symbols by iSAX [10]. This conversion significantly reduces the dimensionality of the problem such that the approximate motif can be computed efficiently. However, their solution has no quality guarantee on the result. Tao et al. [24] propose a locality-sensitive B-tree to answer closest pair query approximately with guarantee for high dimensional objects. Their work is the state-of-the-art solution in this category to the best of our knowledge, but their solution does not guarantee it returns exact motif result.

Regarding *exact* motif discovery, we already discuss two state-of-the-art solutions, MK [5] and SBF [9], in Section II. Besides MK and SBF, Mueen et al. [4] propose a disk-aware algorithm to find exact motif for sequences of million lengths. However, the disk based solution does not fulfill the need of emerging applications (i.e., fast response time). Mueen and Keogh [3] develop a sliding window based motif discovery algorithm which maintains the motif over the most recent history of a stream. Their problem is different from ours where we assume the sequence comes in its entirety per query.

To the best of our knowledge, we are the first work to jointly reduce the number of comparisons and accelerate each individual correlation calculation such that we can boost up the exact motif discovery at scale.

VI. Conclusion

In this paper, we present a scalable framework, Quick-Motif (QM), to handle exact motif discovery efficiently. To the best of our knowledge, we are the first work that discovers motif in a sequence of million lengths in $\sim$20s on a commodity machine (while other approaches take several hours to complete the same discovery task). The performance of our approach enables the possibility to offer online motif discovery in emerging applications.

In the future, we plan to implement Quick-Motif on massively parallel hardware architectures likes GPUs or HPC. Another direction is to study the motif discovery problem based on other distance measurements, such as dynamic time warping with normalization.

Acknowledgement

The authors would like to thank Prof. Eamonn Keogh and Dr. Abdullah Mueen for collecting and publishing the datasets and providing the source code of MK. This project was supported by grant MYRG2014-00106-FST and MYRG105-FST13-GZG from University of Macau RC, grant FDCT/106/2012/A3 from Macau FDCT, and grant GRF 152201/14E from Hong Kong RGC.

References

[1] N. Castro and P. J. Azevedo, "Multiresolution motif discovery in time series," in *SDM*, 2010, pp. 665–676.

[2] Y. Li, J. Lin, and T. Oates, "Visualizing variable-length time series motifs," in *SDM*, 2012, pp. 895–906.

[3] A. Mueen and E. J. Keogh, "Online discovery and maintenance of time series motifs," in *KDD*, 2010, pp. 1089–1098.

[4] A. Mueen, E. J. Keogh, and N. B. Shamlo, "Finding time series motifs in disk-resident data," in *ICDM*, 2009, pp. 367–376.

[5] A. Mueen, E. J. Keogh, Q. Zhu, S. Cash, and M. B. Westover, "Exact discovery of time series motifs," in *SDM*, 2009, pp. 473–484.

[6] E. J. Keogh, J. Lin, and A. W.-C. Fu, "Hot sax: Efficiently finding the most unusual time series subsequence," in *ICDM*, 2005, pp. 226–233.

[7] T. Rakthanmanon, E. J. Keogh, S. Lonardi, and S. Evans, "Time series epenthesis: Clustering time series streams requires ignoring some data," in *ICDM*, 2011, pp. 547–556.

[8] J. Lin, E. Keogh, S. Lonardi, and P. Patel, "Finding motifs in time series," in *Proc. of 2nd Workshop on Temporal Data Mining*, 2002.

[9] A. Mueen, "Enumeration of time series motifs of all lengths," in *2013 IEEE 13th International Conference on Data Mining, Dallas, TX, USA, December 7-10, 2013*, 2013, pp. 547–556.

[10] J. Shieh and E. J. Keogh, "*i*sax: indexing and mining terabyte sized time series," in *KDD*, 2008, pp. 623–631.

[11] Y. Li, L. H. U, M. L. Yiu, and Z. Gong, "Discovering longest-lasting correlation in sequence databases," *PVLDB*, vol. 6, no. 14, pp. 1666–1677, 2013.

[12] T. Rakthanmanon, B. J. L. Campana, A. Mueen, G. E. A. P. A. Batista, M. B. Westover, Q. Zhu, J. Zakaria, and E. J. Keogh, "Searching and mining trillions of time series subsequences under dynamic time warping," in *KDD*, 2012, pp. 262–270.

[13] B. Y. chi Chiu, E. J. Keogh, and S. Lonardi, "Probabilistic discovery of time series motifs," in *KDD*, 2003, pp. 493–498.

[14] E. J. Keogh and S. Kasetty, "On the need for time series data mining benchmarks: a survey and empirical demonstration," in *KDD*, 2002, pp. 102–111.

[15] A. Narang and S. Bhattacherjee, "Parallel exact time series motif discovery," in *Euro-Par (2)*, 2010, pp. 304–315.

[16] J. Lin, E. J. Keogh, L. Wei, and S. Lonardi, "Experiencing sax: a novel symbolic representation of time series," *Data Min. Knowl. Discov.*, vol. 15, no. 2, pp. 107–144, 2007.

[17] B.-K. Yi and C. Faloutsos, "Fast time sequence indexing for arbitrary lp norms," in *VLDB*, 2000, pp. 385–394.

[18] I. Kamel and C. Faloutsos, "On packing r-trees," in *CIKM*, 1993, pp. 490–499.

[19] G. R. Hjaltason and H. Samet, "Incremental distance join algorithms for spatial databases," in *SIGMOD Conference*, 1998, pp. 237–248.

[20] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos, "Closest pair queries in spatial databases," in *SIGMOD Conference*, 2000, pp. 189–200.

[21] M. A. Cheema, X. Lin, H. Wang, J. Wang, and W. Zhang, "A unified approach for computing top-k pairs in multidimensional space," in *ICDE*, 2011, pp. 1031–1042.

[22] "Quick-motif," http://degroup.cis.umac.mo/quickmotifs/.

[23] R. Agrawal, C. Faloutsos, and A. N. Swami, "Efficient similarity search in sequence databases," in *FODO*, 1993, pp. 69–84.

[24] Y. Tao, K. Yi, C. Sheng, and P. Kalnis, "Efficient and accurate nearest neighbor and closest pair search in high-dimensional space," *ACM Trans. Database Syst.*, vol. 35, no. 3, 2010.