Efficient Algorithms for Finding the Closest l-mers in Biological Data

Xingyu Cai, Abdullah-Al Mamun, Sanguthevar Rajasekaran, *Fellow, IEEE*

Abstract—With the advances in the next generation sequencing technology, huge amounts of data have been and get generated in biology. A bottleneck in dealing with such datasets lies in developing effective algorithms for extracting useful information from them. Algorithms for finding patterns in biological data pave the way for extracting crucial information from the voluminous datasets. In this paper we focus on a fundamental pattern, namely, the closest l-mers. Given a set of m biological strings $S_1, S_2, \ldots, S_m$ and an integer l, the problem of interest is that of finding an l-mer from each string such that the distance among them is the least. I.e., we want to find m l-mers $X_1, X_2, \ldots, X_m$ such that X_i is an l-mer in S_i (for $1 \leq i \leq m$) and the Hamming distance among these m l-mers is the least (from among all such possible l-mers). This problem has many applications. An application of great importance is motif search. Algorithms for finding the closest l-mers have been used in solving the (l, d)-motif search problem (see e.g., [1], [2]). In this paper novel exact and approximate algorithms are proposed for this problem for the case of $m > 2$. In particular, a comprehensive experimental evaluation is performed for $m = 3$, along with a further empirical study of $m = 4$ and 5. We also extend our solution to Euclidean distance measurement metric if the sequences contain real numbers.

Index Terms—Closest l-mers; Closest triplet; Efficient algorithms; Randomized algorithms; Time series motifs; (l, d)-motifs

1 Introduction

Large amounts of data get generated in every area of science and engineering. This is especially true in the biological domain. Currently, the bottleneck is not in generating data but is in processing these data. Efficient big data analytics algorithms are called for. A powerful analytics paradigm is patterns finding. In this paper we study an important pattern that can be used to solve many other problems including motif search. Specifically, we investigate the problem of finding the closest l-mers in an input of strings. The biological strings could be DNA sequences, protein sequences, etc. Algorithms for finding the closest l-mers have been used to solve the (l, d)-motif search problem, see for example [1], [2].

The pattern finding problem of interest can be stated as follows. The input are m biological sequences $S_1, S_2, \ldots, S_m$, each of length n, and an integer l. The problem is to find m l-mers $X_1, X_2, \ldots, X_m$ such that X_i is in S_i (for $1 \leq i \leq m$) and the Hamming distance among these l-mers is the least (from out of all such l-mers). X is an l-mer in a sequence S if X is a subsequence of S of length l. Each input sequence can be thought of as a string of characters from a finite alphabet Σ. For instance, each input sequence could be a DNA sequence or a protein sequence. We refer to this pattern finding problem as the closest l-mers problem (CLP). If $X_i = x_1^i x_2^i x_3^i \ldots x_l^i$, for $1 \leq i \leq m$, are any l-mers, then the Hamming distance among them is defined as $\sum_{j=1}^{l} d(x_j^1, x_j^2, \ldots, x_j^m)$ where $d(x_j^1, x_j^2, \ldots x_j^m)$ is zero if all of the characters $x_j^1, x_j^2, \ldots$, and x_j^m are the same; and $d(x_j^1, x_j^2, \ldots x_j^m)$ is 1 otherwise.

The longest common substring (LCS) problem could be viewed as a dual version of CLP. While CLP finds l-mers that are the closest for a given l, LCS finds the length of the longest common substring. Some relevant papers are: [4], [5]. Another related problem is finding the closest pair of points (CP problem). CLP could be viewed as a special case of CP. A number of papers have been written on this problem (see e.g., [6], [7], [8], [9]).

A special case of the CLP when $m = 2$ has been studied in the literature before. For instance, [1] show that this problem can be solved in $O(n^2)$ time for $m = 2$, where n is the length of each of the two input sequences. Note that a trivial algorithm to solve this problem will examine each pair of l-mers A and B where A comes from the first sequence and B comes from the second sequence, compute the Hamming distance between A and B, and output the pair of l-mers with the least distance. This brute force algorithm runs in time $O(n^2 l)$. The $O(n^2)$-time algorithm has been used in solving the (l, d)-motif search problem (see e.g., [1], [2]). Time series motif mining could be viewed as a special case of CLP, and many algorithms have been recently used to solve this problem, such as FFT technique in [10] and $O(n^2)$ methods in [1], [11] [12], and embedding-based approach in [13].

The case of $m > 2$ is very important as well. For instance, in the case of (l, d)-motif search, an algorithm for the case of $m > 2$ can be used in the algorithms of [1], [2] in which case the performance of these algorithms will improve. Also, for the time series motif mining problem, m being more than 2 can provide deeper insights. The problem of time series motif mining can be thought of as that of detecting two events (that occur in two different times) that are very

- *Department of Computer Science and Engineering, University of Connecticut, Storrs, CT, 06269. xingyu.cai@uconn.edu abdullah.am.cs@gmail.com sanguthevar.rajasekaran@uconn.edu.*
- *Xingyu Cai and Abdullah-Al Mamun are co-first authors with equal contribution; Sanguthevar Rajasekaran is the corresponding author.*
- *A preliminary version of this paper was presented in the IEEE International Conference on Bioinformatics and Biomedicine (BIBM), 2017 [3].*

similar to each other. Equally (and perhaps more) important will be the problem of detecting m (> 2) events that are very similar among themselves.

In this paper we present novel algorithms for solving the CLP when $m > 2$. For $m = 3$, we refer to this special case of the CLP as the **closest triplet** problem. Specifically, we offer three different algorithms. Two of these are exact and the third one is approximate. An algorithm is exact if it always outputs the closest l-mers. On the other hand, an approximate algorithm may not output the closest l-mers all the time. In general it outputs l-mers whose distance is very nearly the same as that of the closest l-mers. There is a closely related problem that l-mers could come from the same sequence, and we also extend our algorithms to address this problem, by putting one additional constraint that the l-mers should not overlap. In addition to closest triplet problem, we also extend our algorithms to 4 and 5 input sequences, and provide our experimental results.

Applications: The CLP has many applications. From among these, the (l, d)-motif search is an important problem since motifs can be used to identify transcription factors and their binding sites, composite regulatory patterns, similarity between families of proteins, etc. The (l, d)-motif search (LDMS) problem is stated as follows: Input are n sequences $S_1, S_2, \ldots, S_n$ and integers l and d. The task is to find all the strings M of length l such that M occurs in each of the input sequences within a Hamming distance of d. Each such string M is called an (l, d)-motif. This problem is known to be $\mathcal{NP}$-hard. The WINNOWER algorithm of [1] uses the $O(n^2)$-time closest l-mers algorithm as a crucial step in solving the LDMS problem. In this algorithm they construct a graph $G(V, E)$ where there is a node corresponding to every l-mer in every input sequence. Two nodes are connected by an edge if the Hamming distance between them is no more than $2d$. Followed by the construction of this graph, the algorithm proceeds to look for large cliques in this graph. Each such clique is a candidate for an (l, d)-motif. We can speed up this algorithm as follows: While creating edges in the graph, for every three l-mers x, y, z we form a triangle connecting them if $d(x, y) + d(y, z) + d(z, x) \leq 6d$. No other edges will be included in G. We can find all such triplets efficiently using the CLP algorithms we present in this paper. We can extend this idea further by considering more than three l-mers. The closest l-mers algorithm has also been used in the PMSPrune algorithm of [2] for solving the LDMS problem.

Another important application is that of finding time series motifs. The problem of finding time series motifs can be stated as follows: We are given a sequence S of real numbers and an integer l. The goal is to identify two subsequences of S of length l each that are the most similar (similarity can be defined in various ways, such as Hamming distance, Euclidean distance, etc.) to each other (from among all pairs of subsequences of length l each) [14]. These most similar subsequences are referred to as time series motifs. Yet another application lies in industrial processing. The problem of Control Loop Performance Monitoring (CLPM) is that of identifying related processes [15]. For example, one of the processes might be deviating from its expected behavior and this may be caused by another process. We may want to identify this process. This problem is typically solved by analyzing the time series data from each of the processes and looking for similar subseries. Clearly, there could be (much) more than 2 processes.

A Generalization: When CLP is defined for biological sequences the distance of interest is the Hamming distance. On the other hand, time series data are sequences of real numbers. The distance between two l-mers has to be modified. Several possibilities such as Euclidean distance and Pearson's correlation coefficient have been explored in the literature (see e.g., [12], [14], [16]).

When we extend the CLP for $m > 2$, we have to revisit the notion of distance. When the input has biological sequences, we can continue to used Hamming distance as defined above. If the input consists of time series data, many possibilities arise. Consider the case of $m = 3$. Let X, Y, and Z be any three l-mers. Then, one possible distance among these three is the **pairwise-sum distance** $d(X, Y, Z) = d(X, Y) + d(Y, Z) + d(Z, X)$ where $d(X, Y)$ is the Euclidean distance between X and Y. Hamming distance could be also calculated in a pairwise-sum manner. Thus we refer to the previous definition of Hamming distance as **direct** Hamming distance of a tuple. Other distance metrics are also possible.

Paper Organization: The rest of this paper is organized as follows. In Section 2 we first review existing algorithms for CLP when $m = 2$. This special case is called the *closest pair of subsequence problem*. Next in Section 3, we propose two exact algorithms. The first algorithm uses $O(n^2)$ multiplications and $O(n^3)$ addition operations, and uses $O(n^2)$ memory. We call this algorithm **Exact-0**. The second algorithm has a run time of $O(n^3)$, but only uses $O(1)$ memory. We call the second algorithm **Exact-1**. Another version of the second algorithm takes $O(n)$ memory but reduces the running time to $O(n^3 - n^2 l)$. Note that the second version only applies to pairwise-sum distances. In the subsequent section we present our approximate algorithm, called **Approx**. We show that the run time of this algorithm is $O(n^2 + nKl)$ with a high probability. Here K is a parameter to be chosen in the algorithm. In Section 5 we present our experimental results for $m = 3$. We have used both biological and time series data, and employed direct Hamming distance and pairwise-sum Euclidean distance, respectively. Additional experiments are performed for $m = 4$ and 5 cases in Section 6, to show the robustness of our proposed exact and approximate algorithms. Section 7 provides some concluding remarks and future directions.

2 Background Knowledge

In this section we provide a summary of some basic techniques that have been used to solve the CLP when $m = 2$. An important algorithm in this context is the $O(n^2)$ algorithm proposed by [1]. The early abandoning technique proposed by [14] is also relevant.

2.1 The $O(n^2)$ Time Algorithm of [1]

For solving the closest pair of l-mers problem, Pevzner and Sze exploit the overlaps during the process of computing

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2018.2843364, IEEE/ACM Transactions on Computational Biology and Bioinformatics

pairwise distances. This eliminates the dependence of the run time on l [1]. Let $S = s_1, s_2, \ldots, s_n$ be any given sequence data and let l be the length of the subsequences we are interested in. The problem of finding the closest pair of subsequences in S can be decomposed to $(n-l+1)$ subproblems. Let these subproblems be referred to as $\mathcal{P}_i$, for $1 \leq i \leq (n-l+1)$. Each $\mathcal{P}_i$ computes the distance between the following pairs of subsequences of length l: $([s_j, s_{j+1}, \ldots, s_{j+l-1}], [s_{i+j-1}, s_{i+j}, \ldots, s_{i+j+l-2}])$, for $1 \leq j \leq (n-l+1)$. Note that in these distance calculations, we can ignore any pair if the elements $s_{n'}$ (for $n' > n$) appear in any of the two subsequences. Let the distance between the pair $((s_j, s_{j+1}, \ldots, s_{j+l-1}), (s_{i+j-1}, s_{i+j}, \ldots, s_{i+j+l-2}))$ be d_j^i, for $1 \leq j \leq (n-l+1)$.

[1]'s algorithm makes use of the overlaps in consecutive pairs. We use the Euclidean distance metric as an example here but it is easy to extend our discussion to Hamming distance as well. Since $(d_j^i)^2 = (s_j - s_{i+j})^2 + \ldots + (s_{j+l-1} - s_{i+j+l-1)})^2$, the next pair's squared distance could be expressed as $(d_{j+1}^i)^2 = (s_{j+1} - s_{i+j+1})^2 + \ldots + (s_{j+l} - s_{i+j+l})^2 = (d_j^i)^2 - (s_j - s_{i+j})^2 + (s_{j+l} - s_{i+j+l})^2$.

Clearly, the computation of $(d_1^i)^2$ takes $O(l)$ time. Note that $(d_j^i)^2$ can be obtained from $(d_{j-1}^i)^2$ in an additional $O(1)$ time (for $j > 1$). Thus the problem $\mathcal{P}_i$ can be solved sequentially in a total of $O(n)$ time (for any specific value of i, $1 \leq i \leq (n-l+1)$). Since there are a total of n subproblems, the total running time is $O(n^2)$, which is independent of l. In cases of even moderately large dimensions, e.g., $l = 100$, the speedup over brute-force could be as large as 100 times, which is a non-trivial improvement.

2.2 Early-abandoning Methods in [14]

In [14], the authors proposed an enhanced version of the brute-force algorithm for finding the time series motifs. The techniques they use improve the run time by a large factor, and one of them is the early-abandoning method. Early-abandoning method takes advantage of the fact that the distance is computed as a summation of l elements, or $\sum_{i=1}^{l} d(A_{j+i}, A_{k+i})$, sequentially. If in the middle of the process when the partial sum exceeds the current best-so-far, which is an upper bound of the closest distance, then we can immediately stop the computation of the distance between the current pair. This method is very useful in practice and will be also employed in our approximate algorithm.

3 The Exact Algorithms

When $m = 3$ we can solve the CLP in $O(n^3 l)$ time in a straight forward way. The idea is to compute the distance among every triplet of l-mers. For each triplet the time spent is $O(l)$ and there are $O(n^3)$ triplets.

3.1 Exact-0 Algorithm for Pairwise-sum Distances

We can solve the CLP for $m = 3$ in $O(n^3)$ time using the algorithm of [1] as a subroutine. This algorithm will work as follows: 1) Use the algorithm of [1] to compute pairwise distances in $O(n^2)$ time. Store all of these distances. Followed by this, compute the distance for each possible triplet of l-mers. Note that the distance for any triplet can be computed in $O(1)$ time (since the pairwise distances are available). For instance if (X, Y, Z) is the triplet under concern, its distance is $d(X,Y) + d(Y,Z) + d(Z,X)$ and the distances $d(X,Y), d(Y,Z)$, and $d(Z,X)$ have already been computed and are available. Since there are $O(n^3)$ triplets, the total addition operations will be $O(n^3)$. Note that for this algorithm we need $O(n^2)$ space. We get the following Theorem:

Theorem 3.1. We can use Exact-0 algorithm to solve the CLP for $m = 3$ using $O(n^2)$ multiplications and $O(n^3)$ addition operations, as well as $O(n^2)$ space. □

3.2 Exact-1 Algorithm

If the input size n is large, the $O(n^2)$ memory cost may be prohibitive. For example, when $n = 40 \times 10^3$, using double precision storage, the algorithm would require roughly 10 GB of memory. This is quite large. Besides, as memory usage increases, the memory accessing cost will become dominant and make the algorithm take longer time to finish.

Motivated by this, we have developed a memory efficient algorithm that solves this problem in $O(n^3)$ time with only a constant memory requirement. In the case of pairwise-sum distance measurement, $O(n^2 l)$ time could be saved at the cost of $O(n)$ memory. We thus have two versions: The first version takes $O(n^3)$ time and uses $O(1)$ memory; the second version takes $O(n^3 - n^2 l)$ time and employs $O(n)$ memory. The second version is very useful when l is not far less than n. For instance, if $l = 0.3n$, then 30% of the total running time could be reduced.

3.2.1 *Version 1:* $O(1)$ *Memory*

The key idea to reduce the memory cost from $O(n^2)$ to $O(1)$, is by exploiting the overlaps like in [1]. Rather than using [1]'s algorithm as a subroutine to compute all pairwise distances in the first step, we split the entire procedure into subproblems $\mathcal{P}_{ik}$ such that each subproblem represents a unique alignment (i, k) and outputs distances of the triplets $(a, a+i, a+i+k), a \in [1, n]$. Clearly, consecutively outputting the distance as a shifts, would cost $O(n)$ time for each subproblem, and there' are a total of $O(n^2)$ subproblems. So the total running time for this algorithm is $O(n^3)$. Besides, since only one set of distances (for pairwise-sum distance, three pairwise distances are stored; for direct distance, one triplet distance is stored) needs to be stored in memory, the memory cost becomes $O(1)$ during the entire process. This can be seen as an enhanced version of [1]'s algorithm. We arrive at the following Theorem:

Theorem 3.2. The CLP can be solved in $O(n^3)$ time using $O(1)$ space applying Exact-1 algorithm version 1. □

3.2.2 *Version 2:* $O(n)$ *Memory*

Both Exact-0 and Exact-1 constant memory version are two extreme cases and we are seeking one in the middle by using an affordable amount of memory to reduce the computation time. This could be achieved in the case of pairwise-sum distance metric, because the pairwise distances that have been calculated could be partially stored instead of fully storing (as in Exact-0).

Without out loss of generality, we give an illustration using the example of finding the closest 3 l-mers from one

single sequence under pairwise-sum measurement metric, with a constraint that there are no overlaps for l-mers in the closest triplet. In the previous $O(1)$ version, for each alignment $< i, k >$, the starting cost to compute $d(0,i), d(i,i+k), d(0,i+k)$ still requires $O(l)$ time each. And since there are $O(n^2)$ alignments, the subproblems' starting costs accumulate to $O(n^2 l)$. After starting, all the remaining distances are calculated in only $O(1)$ time. As a result, removing the starting cost could save a decent fraction of the total running time. As noticed, the majority of starting cost is in the form of $d(0,i), i \in [l, n-2l]$. Thus a simple solution is to store these values in memory to avoid repetition in computing them. This only require $O(n)$ storage and the running time is reduced to $O(n^3 - n^2 l)$ as a consequence.

Algorithm 1 Exact-1 Algorithm with $O(n)$ Memory

Input: Sequence $A = s_1, s_2, \ldots, s_n$; subsequence A_t is defined as $A_t = [s_t, s_{t+1}, \ldots, s_{t+l-1}]$; $\hat{d}(A_{t_1}, A_{t_2})$ denotes **squared** Euclidean distance between A_{t_1}, A_{t_2}

Output: A triplet of subsequences that has the least pairwise-sum Euclidean distance

Set best-so-far $b = \infty$
for $i = 0$ to $n - l$ **do**
  Compute and store $D_1[i] = \hat{d}(A_0, A_i)$
end for
for $k = l$ to $n - l$ **do**
  Obtain $\hat{d}_1 \leftarrow D_1[k]$
  for $j = l$ to $n - k$ **do**
    Compute $\hat{d}_2 = \hat{d}(A_k, A_{k+j})$; Obtain $\hat{d}_3 \leftarrow D_1[k+j]$
    $tmp = \sqrt{\hat{d}_1} + \sqrt{\hat{d}_2} + \sqrt{\hat{d}_3}$
    if $tmp < b$ **then**
      update $b \leftarrow tmp$ and the corresponding indices
    end if
    for $i = 0$ to $n - l - k$ **do**
      $\hat{d}_1 = \hat{d}_1 - (s_i - s_{i+k})^2 + (s_{i+l} - s_{i+k+l})^2$
      $\hat{d}_2 = \hat{d}_2 - (s_{i+k} - s_{i+k+j})^2 + (s_{i+k+l} - s_{i+k+j+l})^2$
      $\hat{d}_3 = \hat{d}_3 - (s_i - s_{i+k+j})^2 + (s_{i+l} - s_{i+k+j+l})^2$
      $tmp = \sqrt{\hat{d}_1} + \sqrt{\hat{d}_2} + \sqrt{\hat{d}_3}$
      if $tmp < b$ **then**
        update $b \leftarrow tmp$ and the corresponding indices
      end if
    end for
  end for
end for
return b and the corresponding indices

Details of this algorithm are given in Algorithm 1. Note that the problem of finding 3 closest l-mers among 3 separate sequences (A, B, C) can be solved using Algorithm 1 by removing the non-overlapping constraint, and storing 2 sets of distances $d(A_0, B_i), d(A_i, B_0), i \in [0, n]$. Each set needs $O(n)$ memory and hence only $O(n)$ memory is required in total. We obtain the following Theorem:

Theorem 3.3. We can solve the CLP in $O(n^3 - n^2 l)$ time using $O(n)$ memory applying Exact-1 algorithm version 2. □

4 An Approximate Algorithm: Approx

The brute-force algorithm for $m = 3$ takes $\mathcal{O}(n^3 l)$ time, which is very large even for moderately large values of n and l. The $\mathcal{O}(n^3)$ algorithms take significantly less time by removing the l factor. Still it takes hours to compute the required triplet from a time series or genome sequence of length 20,000. Therefore it will take days or months to solve the CLP when n is a million or more. To address this problem we have developed a fast approximate algorithm, which has a running time of $\mathcal{O}(n^2 + nKl)$ with a high probability, where K is a user defined parameter.

4.1 Description

Our approximate algorithm works in two phases. In the first phase, the algorithm computes pairwise distances among all possible l-mers and keeps K edges which have the smallest distances. An edge here refers to a pair of l-mers. A priority queue Q is used to identify the best K edges efficiently. To reduce the number of edges that will be inserted into Q, an upper bound on the distance between the closest pair of l-mers is first obtained using random sampling. During initial random sampling, we pick s edges randomly. In each pick, each possible edge has an equal probability. We compute the distance of each edge in the sample and identify the edge with the least distance. Let the distance of this edge be δ_s. We use δ_s as the threshold for edges for inserting them into Q.

The K edges that are in Q, after processing all possible edges, will be used in the second phase. We form candidate triplets as follows: For each of the edges in Q form triplets with every l-mer in the input sequence A. From out of all of these candidate triplets identify and output the one with the least distance. Algorithm 2 shows the details of the algorithm for single sequence version (l-mers come from single sequence). The multi sequences CLP could be easily solved by removing the overlapping constraint.

4.2 Analysis

We choose a random sample of s pairs of l-mers from the input sequence A. The algorithm takes $\mathcal{O}(sl)$ time to calculate pairwise distances of these pairs. In our implementation we choose $s = \Theta(n)$. We get our threshold value δ_s by finding minimum of these distance values. Then we compute the distance between every pair of l-mers of the input sequence A. This can be done in $\mathcal{O}(n^2)$ time. From out of these, we identify the K least distances. Identification of these K pairs is done using a priority queue Q. We insert any pair into Q only if its distance is less than δ_s. Q will have at most K pairs at any time. For each pair that enters Q another pair may have to be deleted. An important question is how many pairs will enter Q in the worst case. We claim that the number of pairs that will enter Q is $O\left(\frac{N}{s}\log n\right)$ with a high probability, where N is the number of possible pairs of l-mers. (Note that $N = O\left(\binom{n}{2}\right)$). This can be proven as follows.

By high probability we mean a probability of $\geq (1 - n^{-\alpha})$, where α is a probability parameter typically assumed to be ≥ 1 (see e.g., [17]). Let G stand for the set of pairs of l-mers of A with the least distances, where $|G| = q$. I.e., q

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2018.2843364, IEEE/ACM Transactions on Computational Biology and Bioinformatics

Algorithm 2 Approx algorithm

Input: A sequence $A = s_1, s_2, \ldots, s_n$, integer l, priority queue Q of size K

Output: A triplet of l-mers whose members that have the least distance.

1: Choose randomly s pairs of l-mers (edges) from A
2: Compute the distance for each pair in the sample, and identify the closest pair in the sample; let this closest distance be δ_s
3: **for** All pairs in A **do**
4: Compute the distance between each pair of l-mers
5: **if** Any pair's distance $< \delta_s$ **then**
6: Push into Q
7: **end if**
8: **end for**
9: Set best triplet-distance as $b = \infty$
10: **for** each l-mer u **do**
11: **for** each pair (v, w) in priority queue **do**
12: Compute distances of (u, v), (u, w) pairs
13: Set triplet distance TD = distance(u, v) + distance(v, w) + distance(u, w)
14: **if** $TD < b$ **then**
15: $b = TD$, update corresponding indices to (u, v, w)
16: **end if**
17: **end for**
18: **end for**
19: **return** b with associated indices

pairs with the least distances are in G. (The value of q will be fixed soon). Let the pairs in G be $p_1, p_2, \ldots, p_q$. Probability that p_1 is in the random sample is $\frac{s}{N}$. Probability that p_1 is not in the sample is $1 - \frac{s}{N}$. This means that the probability that none of G is in the sample is $\left(1 - \frac{s}{N}\right)^q$. This probability is $\leq \exp\left(\frac{-sq}{N}\right)$ using the fact that $(1-x)^{1/x} \leq \frac{1}{e}$ when $0 < x < 1$. This probability will be $\leq N^{-\alpha}$ when $q \geq \alpha \frac{N}{s} \log_e N$. This in turn means that at most $\alpha \frac{N}{s} \log_e N \left(= O\left(\frac{N}{s} \log n\right)\right)$ pairs will ever enter Q with a probability of $\geq 1 - N^{-\alpha} \geq (1 - n^{-\alpha})$.

The above analysis implies that the total time spent in maintaining Q is $O\left(\frac{N}{s} \log^2 n\right) = O\left(\frac{n^2}{s} \log^2 n\right)$ with a high probability. Also, steps 1 and 2 in Algorithm 2 take a total of $O(sl)$ time. Step 4 can be done in $O(n^2)$ time. As we have shown before, steps 5 and 6 take a total of $O\left(\frac{n^2}{s} \log^2 n\right)$ time with a high probability. The **for** loop of Step 8 takes $O(nKl)$ time. Therefore, the total run time of Algorithm 2 is $O\left(sl + n^2 + \frac{n^2}{s} \log^2 n + nKl\right)$ with a high probability. If $s = \Theta(n)$, this run time becomes $O(n^2 + nKl)$. Thus we arrive at the following Theorem.

Theorem 4.1. The run time of Approx is $O(n^2 + nKl)$ with a high probability. □

5 Experimental Evaluation

In this section, we evaluate our proposed algorithms for run time and/or accuracy using two existing datasets. Each dataset is tested using one measurement metric (direct Hamming distance and pairwise-sum Euclidean distance). The test platform we are using is equipped with Intel Xeon CPU @ 2.67GHz.

5.1 Genome Dataset

We have performed intensive experiments on human genome data set [18]. We chose 21 chromosomes and grouped them into 7 files each having 3 chromosome sequences. We have run Exact-1 and Approx algorithms in order to identify the closest l-mers among three sequences in each set, and there are 7 sets of genome sequences. For Approx algorithm, we set the priority queue size as $K = n$ to store the top n pairs of candidates as a default. We have used different values for n ranging from 4,000 to 60,000. The first n elements of the 7 sets of genome sequences are used to form the input sequences. Direct Hamming distance is used as the distance metric. For a fixed n and l, we call such a combination a test group, and the running time is calculated as an average over the 7 sets (6 runs each set, a total of 42 runs) of genome sequences for this group. We report the accuracy of Approx using the Hit Rate, which measures how many times out of 42 runs, the closest l-mers are identified.

At first we compare our proposed algorithms with the ($O(n^3 l)$ time) brute-force algorithm. The result is given in Table 1. From the table we see that the brute-force algorithm performs worse even when $n = 4,000$. Thus in later experiments we will not include the brute-force algorithm and only compare our proposed approaches.

TABLE 1
Running time comparison with the Brute-force algorithm

$n = 4,000$	l=100	l=200	l=300	l=400	l=500
Approx	4.2	5.6	7.6	9.4	7.7
Exact	207.0	215.1	212.3	203.0	199.0
Brute-force	8,353.0	19,613.0	28,744.6	37,178.9	44,607.9

The next experiment provides a full evaluation when n ranges from $4,000$ to $10,000$, with $l = 100$ to 500. The running time of Exact-1 and Approx are provided in Semilog-Y plot for a better illustration in Figure 1. From the plot, we clearly see that the Approx algorithm outperforms Exact-1 by more than one order of magnitude for all the 5 different l values. Also, as the dataset size n increases, the running time difference increases.

Next we want to investigate how the performance changes as l varies, for a fixed n. We choose $n = 6,000$ and change l from 100 to 500. The running times are shown in Figure 2. The upper plot shows the running time for the Approx algorithm and the lower one represents Exact-1 algorithm. The observation here is that since Approx algorithm's running time depends on l, as l increases, the running time of Approx slightly increases. On the contrary, Exact-1 algorithm is dimension free. Note that the exact number of triplets is $(n - l)^3$ rather than the asymptotic value n^3. If l increases, the number of triplets would actually decrease a little bit, thus the Exact algorithm's running time slightly decreases. But still, Approx is much faster than Exact-1 even when $l = 500$.

The speedups for all n and l combinations are given in Figure 3. Speed up is defined as the running time of Exact-1 divided by the running time of Approx algorithm. As

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2018.2843364, IEEE/ACM Transactions on Computational Biology and Bioinformatics

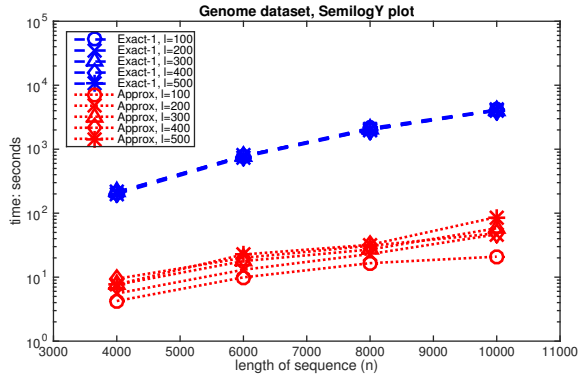

Fig. 1. Running time comparison on Genome dataset

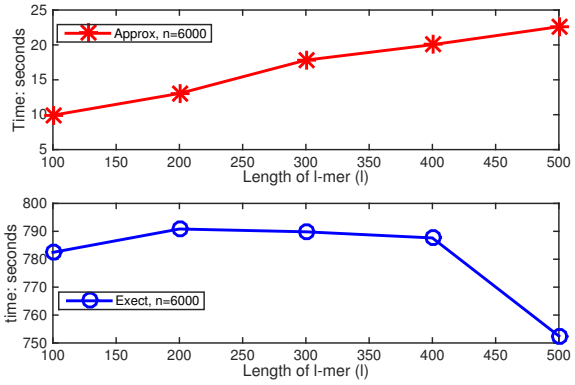

Fig. 2. Running time vs l

can be seen, speedup decreases as l grows, and increases as n grows, which matches our expectation and theoretical analysis.

In the last part of time comparison, we are testing our algorithm on large datasets with $n = 10,000,\ 20,000,\ 40,000,\ 60,000$ and $l = 200,\ 400,\ 600,\ 800,\ 1,000$. Using this setting, Approx could output the results within two hours, while Exact-1 exceeds our experimental limit of 15 hours for $n = 40,000$ and above. The details of running time for large datasets are given in Table 2.

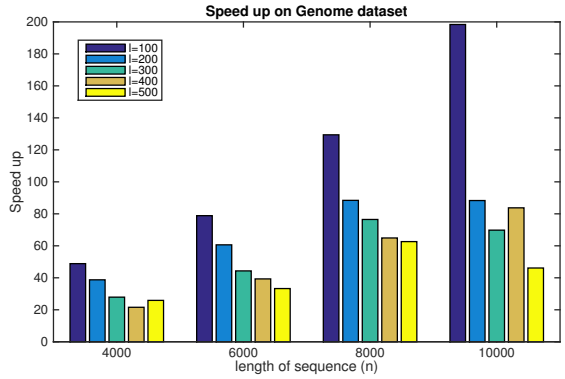

Fig. 3. Speed up of Approx algorithm

TABLE 2
Running times for large datasets

	l=200	l=400	l=600	l=800	l=1,000
n			Approx		
10,000	32.2	61.2	92.9	86.0	112.3
20,000	125.3	231.6	275.8	460.1	493.9
40,000	594.2	935.2	1,452.4	1,676.7	2,110.0
60,000	961.0	2,126.5	3,202.2	3,880.6	5,307.8
n			Exact-1		
10,000	46.4	58.1	66.8	95.9	121.6
20,000	37,777.8	37,771.4	37,500.4	36,252.0	36,567.0
40,000	NA	NA	NA	NA	NA
60,000	NA	NA	NA	NA	NA

Since time and accuracy are trade-offs for an approximate algorithm, we also investigate the output accuracy of Approx with a default setting of $K = n$. The result is given in Table 3 and Table 4. The hit rate is defined as the number of times identifying the closest distance, divided by the total running times. From the table we clearly see that the accuracy of Approx algorithm is very high. Especially for larger sequence lengths such as $n = 10,000$, most of the times Approx could achieve very good hit rate for different l-mer lengths.

TABLE 3
Approx's hit rate for small n and l values ($K = n$)

n	l=100	l=200	l=300	l=400	l=500
4,000	0.55	0.55	0.69	0.79	0.50
6,000	0.57	0.55	0.48	0.43	0.62
8,000	0.81	0.86	0.76	0.48	0.64
10,000	0.98	0.95	0.86	0.69	0.79

TABLE 4
Approx's hit rate for large n and l values ($K = n$)

n	l=200	l=400	l=600	l=800	l=1,000
10,000	0.95	0.69	0.98	0.83	0.79
20,000	0.83	0.86	0.81	0.76	0.74

As K value would affect the time and accuracy of the Approx algorithm, we conduct another experiment to test how the performance changes as K varies. We vary K from $0.001n$ to $10n$, and measure both the accuracy and running time when $n = 10,000$ with different l values. The Hit rate against K value is provided in the first plot of Figure 4. Figure 5's left plot illustrates the running time as K changes. We can easily see that as K increases, both the hit rate and run time increase, for all different l values. This is because small K means a small priority queue and hence the time to maintain Q decreases. As a result, only storing a small number of candidate pairs could reduce the chance of identifying the true closest triplet. From the figures we observe that if $K = 0.1n$, the total run time could be reduced by a large factor, while still maintaining a good accuracy.

5.2 Human Activity Dataset

In this experiment, we evaluate our algorithms under the pairwise-sum distance measurement using Euclidean distance, i.e. $d(A_i, A_j, A_k) = d(A_i, A_j) + d(A_j, A_k) + d(A_i, A_k)$.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2018.2843364, IEEE/ACM Transactions on Computational Biology and Bioinformatics

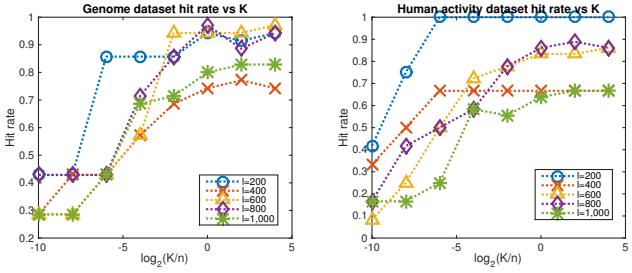

Fig. 4. Hit rate for different K values, $n = 10,000$

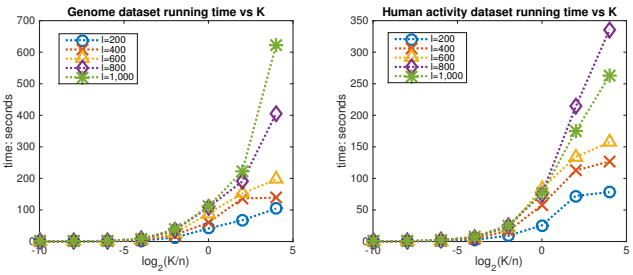

Fig. 5. Running time for different K values, $n = 10,000$

The goal is to identify 3 l-mers from one single sequence A, such that their pairwise-sum distance is minimum, under the constraint that they do not overlap with each other.

The dataset we use is from UCI Machine Learning Repository [19]. For a fair comparison, we have randomly selected one dataset which happens to be the Heterogeneity Activity Recognition Data Set [20]. This contains around 1×10^7 real numbers. This dataset includes cellphone accelerometer and gyroscope recorded data for human activity. There are 6 sensor coordinates in total and each forms a long sequence of numbers.

To perform evaluations, we downsampled the dataset with an interval of 10 for each sequence, and then applied a shifting of 0 and 5 to obtain a total of 12 downsampled sequences. We have performed evaluations on different n and d values. The first n elements in each sequence have been pulled out to form a group of data sequences. The evaluation is based on average performance across 12 data sequences (5 runs per sequence) in each group, and accordingly the accuracy is reported as the Hit Rate (number of Hits out of 60 runs). Three algorithms are evaluated on this dataset, which are Exact-0, Exact-1 and Approx. For Approx, we still set $K = n$ as a default.

Similar to previous experiment, we first inspect the running time of all the three algorithms under different settings of n and l combinations. As shown in Figure 6, three clusters of curves represent three algorithms, respectively. Among them, Approx still runs the fastest and Exact-1 is the most time consuming. Between Approx and Exact-1, Exact-0 gives a moderate running time at the cost of $O(n^2)$ memory. For datasets up to $n = 10,000$, around 600MB memory is occupied by Exact-0. However, since Exact-0 is five times faster than Exact-1, it is very competitive on small to moderate datasets.

Also, the speedup of Approx against both Exact-0 and Exact-1 are given in Figure 7. In the figure, "Speedup 1" and

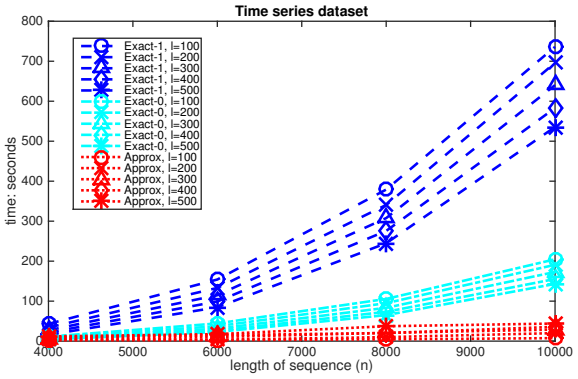

Fig. 6. Running time comparison on Activity dataset

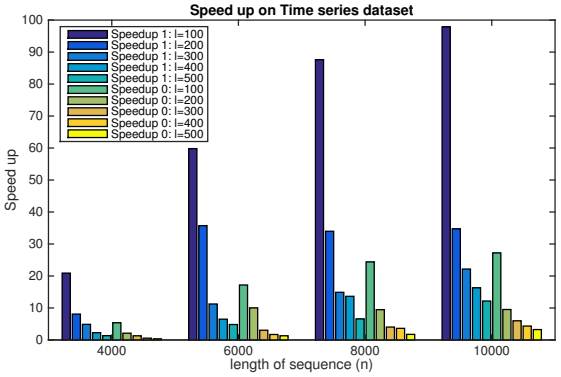

Fig. 7. Speedup against both the exact algorithms

"Speedup 0" represent Approx's speedup over Exact-1 and Exact-0, respectively. The observation is that for small dimensions (l), and for large sequence lengths n, much higher speedup could be achieved using Approx algorithm. For large l values, the running time of Approx could be almost the same as Exact-0, such that Exact-0 could be preferred in these cases as it is an exact algorithm that always outputs the correct answer. However, as stated above, if n is large, Exact-0 may no longer be applicable due to the huge memory cost.

In the next test we demonstrate how these three algorithms' running time varies as l changes. Using the same setting as in the previous experiment, we pick $n = 6,000$ and change l from 100 to 500. Figure 8 plots three curves representing three the algorithms, respectively. As expected, for both exact algorithms Exact-0 and Exact-1, the running time decreases as l increases, because the actual number of triplets $(n - l)^3$ decreases. For Approx, the running time increases due to its dependence on l.

The next experiment is performed on larger n and l values. In particular, $n = 10,000,\ 20,000$, d ranges from 200 to $2,000$. From Table 5, we see that Approx is more than 10 times faster than Exact-0 and 100 times faster than Exact-1 on small l values. For larger l values, Approx is still more than twice faster than Exact-0 and 20 times faster than Exact-1.

Besides the running time performance, we also provide accuracy (# Hits) for all the above tests in Table 6. Due to the fact that K is set depending on n ($K = n$ in the above tests),

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2018.2843364, IEEE/ACM Transactions on Computational Biology and Bioinformatics

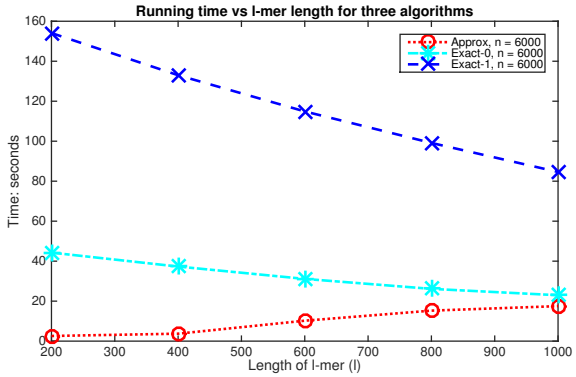

Fig. 8. Running time vs l

TABLE 5
Running times on large datasets

n		l=200	l=400	l=600	l=800	l=1,000
10k	Approx	21.9	46	55.7	74.3	68.5
	Exact-0	167.5	139.4	108.2	88.4	69.0
	Exact-1	695.9	587.8	489.8	403.5	329.4
n		l=400	l=800	l=1,200	l=1,600	l=2,000
20k	Approx	203.6	308.0	497.6	492.0	500.1
	Exact-0	1,404.0	1,146.8	953.4	725.6	582.6
	Exact-1	5,583.9	4,742.6	3,997.6	3,247.1	2,681.1

the accuracy of Approx is consistent for all the different n values, making this a fair experiment for running time comparison. (Approx's running time also depends on K). As shown in Table 6, using a default $K = n$, we are able to obtain a very high accuracy. We get full Hits (out of 60 runs) for $l = 100$ and many other settings. One interesting fact is that as l increases, # Hits decreases slightly. The reason behind this is that since l increases, there are more elements contributing to the total distance computation (note that the distance is a summation of l elements), such that the randomness increases a lot. So there are higher chances that the 3 pairs $(a, b), (b, c), (a, c)$ within the closest triplet (a, b, c) might not exist in the top K closest pairs, but still making the triplet closest.

TABLE 6
Hit rate for Approx, $K = n$

n	l=100	l=200	l=300	l=400	l=500
4,000	0.98	0.98	0.73	0.88	0.67
6,000	1.00	1.00	0.83	0.87	0.67
8,000	1.00	0.98	0.88	1.00	0.77
10,000	1.00	0.98	0.98	0.67	0.97
n	l=200	l=400	l=600	l=800	l=1,000
10,000	0.98	0.95	0.80	0.90	0.62
20,000	1.00	1.00	0.82	0.50	0.75

At the end, we are inspecting how Approx performs if K changes. The Hit rate and the running time are provided in the second plots of Figure 4 and Figure 5. The figure shows similar trends as in the Genome dataset.

5.3 Summary of Experimental Evaluation

In this section we have performed comprehensive evaluations on Genome dataset and Activity dataset. The measurement metrics we used are direct Hamming distance and pairwise-sum Euclidean distance. For Genome dataset, two algorithms Exact-1 and Approx are tested; for Activity dataset, three algorithms Exact-0, Exact-1 and Approx are compared. The experiments are carried on different n and l values. Most of the experiments for Approx are using $K = n$ as a default. We have the following observations:

- The performances are consistent using both measurement metrics on two different datasets, showing our proposed algorithms are robust.
- Exact-0 algorithm runs faster than Exact-1, at a cost of $O(n^2)$ memory. On small datasets, it is very competitive.
- Exact-1 is performing much better than brute-force, making it a good candidate for exact algorithm that always output correct answer.
- Exact algorithms' running times decrease as l increases, while Approx's running time increases
- Approx runs much faster than both the exact algorithms, while maintaining a very high accuracy
- Approx's running time almost increases quadratic or even less on n, while exact algorithms grow cubic on n
- As l increases, the accuracy of Approx slightly decreases, the running time of Approx slightly increases.
- K can be set to $0.1n$ to achieve even better speedups while maintaining a similar accuracy

6 Closest l-mers in More Sequences

Finding closest l-mers from three different sequences (namely, closest triplet problem) is a crucial problem in biological data analysis. There is a further need for developing efficient algorithms for identifying closest l-mers from even more sequences. However, as the number of input sequences increases, the computation time could increase exponentially (especially for exact algorithms).

In this paper, we extend our exact and approximate algorithms to the problem of more than three sequences. In particular, we run our experiments on $m = 4$ and $m = 5$ cases, and report our results in this section. Exact-1 and Approx are evaluated on the genome dataset, and the Hamming distance (direct distance, refer to Section 1 for definition) is adopted as the similarity measurement. To extend Exact-1 (version 1) to handle 4 input sequences, we simply modify it to check the subsequences starting at position $(a, a + i, a + i + j, a + i + j + k)$ for a specific alignment (i, j, k) (recall that for three sequences, we check the alignment of (i, j)). The same modification is applied when handling 5 input sequences, but adding one more variable into the alignment tuple. The Approx algorithm also works in a similar way as the 3-input-sequences case. It maintains a priority queue of size K to store the closest pairs from two input sequences. When collecting the distances, it checks all the pairs in the queue, against every l-mer from all the other input sequences. Note that the proposed algorithms can be easily be extended to more sequences, but the running time would also increase. Parallelization could alleviate the time concern by taking advantage of multi-core systems, and our proposed approaches are also easily parallelizable.

In this section, we provide a brief experimental comparison of our proposed exact and approximate algorithms. When the input has 4 sequences, we set $n = 1000$ and vary l from 10 to 50. When there are 5 input sequences, we set $n = 200$ and change l from 6 to 10. The average running time of Exact-1 algorithm and Approx algorithm are reported in Table 7. Also, the hit rate of our Approx algorithm is provided in Table 8 to show the accuracy. Note that K value is set to be n in both of the experiments. We did not employ Exact-0 in this experiment because of the large memory cost, making it not applicable in most common computing platforms.

TABLE 7
Running times for 4 and 5 input sequences

m=4, n=1,000	l=10	l=20	l=30	l=40	l=50
Approx	21.8	59.8	95.5	109.9	154.8
Exact-1	10,707.1	10,478.5	10,298.7	10,102.8	9,958.1
m=5, n=200	l=6	l=7	l=8	l=9	l=10
Approx	13.7	15	16.3	17.4	18.6
Exact-1	6,957.6	6,921.4	6,769.8	6,739.7	6,679.6

TABLE 8
Hit rate for 4 and 5 input sequences

m=4	n=1,000	l=10	l=20	l=30	l=40	l=50
	Approx	0.86	0.84	0.79	0.7	0.74
m=5	n=200	l=6	l=7	l=8	l=9	l=10
	Approx	1.00	1.00	1.00	0.99	0.91

From the table we can see that the running time of Approx is much better than that of Exact-1, where the brute-force cannot finish for even small n and l values. For example, Exact could take more than 2 hours, and Approx only takesless than 20 seconds when $n = 200, l = 10$ for 5 input sequences. This demonstrates that the Approx algorithm is especially suitable for large number of input sequences.

7 Conclusions

In this paper we consider the problem of finding the closest l-mers when the input has multiple sequences. We offer exact and approximate algorithms, under direct and pairwise-sum distance measurement metrics. All of our algorithms have been implemented and evaluated on both real biological datasets and time series sequences. Our experimental results reveal that our exact algorithms are much more efficient than the brute-force algorithm. Our approximate algorithm outperforms the exact algorithms with a speedup of more than 100 keeping a very good accuracy. For the case of more than three sequences, in particular $m = 4$ and 5, we also have tested both of our Exact-1 and Approx algorithms on the biological datasets. Experimental results show the strong feasibility of our approximate algorithms that can handle multiple input sequences of non-trivial lengths. In future we plan to consider the case of even larger m values. We will also focus on improving the performance of the algorithms we have presented in this paper.

Acknowledgment

This work has been supported in part by the NSF grant 1447711 and 1743418.

References

[1] P. A. Pevzner, S.-H. Sze, *et al.*, "Combinatorial approaches to finding subtle signals in dna sequences.," in *ISMB*, vol. 8, pp. 269–278, 2000.

[2] J. Davila, S. Balla, and S. Rajasekaran, "Fast and practical algorithms for planted (l, d) motif search," *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, vol. 4, no. 4, pp. 544–552, 2007.

[3] X. Cai, A.-A. Mamun, and S. Rajasckaran, "Novel algorithms for finding the closest l-mers in biological data," in *Bioinformatics and Biomedicine (BIBM), 2017 IEEE International Conference on*, IEEE, 2017.

[4] A. Gorbenko and V. Popov, "On the longest common subsequence problem," *Applied Mathematical Sciences*, vol. 6, no. 116, pp. 5781–5787, 2012.

[5] C.-A. Leimeister and B. Morgenstern, "Kmacs: the k-mismatch average common substring approach to alignment-free sequence comparison," *Bioinformatics*, vol. 30, no. 14, pp. 2000–2008, 2014.

[6] T. J. W. I. R. C. R. Division and M. Rabin, *Probabilistic algorithms*. 1976.

[7] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos, "Closest pair queries in spatial databases," in *ACM SIGMOD Record*, vol. 29, pp. 189–200, ACM, 2000.

[8] P. Indyk, M. Lewenstein, O. Lipsky, and E. Porat, "Closest pair problems in very high dimensions," in *ICALP*, vol. 3142, pp. 782–792, Springer, 2004.

[9] Y. Tao, K. Yi, C. Sheng, and P. Kalnis, "Efficient and accurate nearest neighbor and closest pair search in high-dimensional space," *ACM Transactions on Database Systems (TODS)*, vol. 35, no. 3, p. 20, 2010.

[10] Y. Li, M. L. Yiu, Z. Gong, *et al.*, "Quick-motif: An efficient and scalable framework for exact motif discovery," in *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pp. 579–590, IEEE, 2015.

[11] C.-C. M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H. A. Dau, D. F. Silva, A. Mueen, and E. Keogh, "Matrix profile i: All pairs similarity joins for time series: A unifying view that includes motifs, discords and shapelets," in *IEEE ICDM*, 2016.

[12] Y. Zhu, Z. Zimmerman, N. S. Senobari, C.-C. M. Yeh, G. Funning, A. Mueen, P. Brisk, and E. Keogh, "Matrix profile ii: Exploiting a novel algorithm and gpus to break the one hundred million barrier for time series motifs and joins," in *Data Mining (ICDM), 2016 IEEE 16th International Conference on*, pp. 739–748, IEEE, 2016.

[13] P. Papapetrou, V. Athitsos, M. Potamias, G. Kollios, and D. Gunopulos, "Embedding-based subsequence matching in time-series databases," *ACM Transactions on Database Systems (TODS)*, vol. 36, no. 3, p. 17, 2011.

[14] A. Mueen, E. Keogh, Q. Zhu, S. Cash, and B. Westover, "Exact discovery of time series motifs," in *Proceedings of the 2009 SIAM International Conference on Data Mining*, pp. 473–484, SIAM, 2009.

[15] M. Bauer, A. Horch, L. Xie, M. Jelali, and N. Thornhill, "The current state of control loop performance monitoring–a survey of application in industry," *Journal of Process Control*, vol. 38, pp. 1–10, 2016.

[16] B. Chiu, E. Keogh, and S. Lonardi, "Probabilistic discovery of time series motifs," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 493–498, ACM, 2003.

[17] E. Horowitz, S. Sahni, and S. Rajasckaran, *Computer algorithms: C++*. WH Freeman & Co., 1996.

[18] K. D. Pruitt, G. R. Brown, S. M. Hiatt, F. Thibaud-Nissen, A. Astashyn, O. Ermolaeva, C. M. Farrell, J. Hart, M. J. Landrum, K. M. McGarvey, *et al.*, "Refseq: an update on mammalian reference sequences," *Nucleic acids research*, vol. 42, no. D1, pp. D756–D763, 2013.

[19] A. Asuncion and D. Newman, "Uci machine learning repository," 2007.

[20] A. Stisen, H. Blunck, S. Bhattacharya, T. S. Prentow, M. B. Kjærgaard, A. Dey, T. Sonne, and M. M. Jensen, "Smart devices are different: Assessing and mitigatingmobile sensing heterogeneities for activity recognition," in *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pp. 127–140, ACM, 2015.