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

Embedding-based Subsequence Matching with Gaps-Range-Tolerances: a Query-By-Humming application

Alexios Kotsifakos · Isak Karlsson · Panagiotis Papapetrou · Vassilis Athitsos · Dimitrios Gunopulos

Received: date / Accepted: date

Abstract We present a subsequence matching framework that allows for gaps in both query and target sequences, employs variable matching tolerance efficiently tuned for each query and target sequence, and constrains the maximum matching range. Using this framework, a dynamic programming method is proposed, called SMBGT, that, given a short query sequence Q and a large database, identifies in quadratic time the subsequence of the database that best matches Q. SMBGT is highly applicable to music retrieval. However, in Query-By-Humming applications, runtime is critical. Hence, we propose a novel embedding-based approach, called ISMBGT, for speeding up search under SMBGT. Using a set of reference sequences, ISMBGT maps both Q and each position of each database sequence into vectors. The database vectors closest to the query vector are identified, and SMBGT is then applied between Q and the subsequences that correspond to those database vectors. The key novelties of ISMBGT are that it does not require training, it is query-sensitive, and it exploits the flexibility of SMBGT. We present an extensive experimental evaluation using synthetic and hummed queries on a large music database. Our findings show that ISMBGT can achieve speedups of up to an order of magnitude against brute-force search and over an order of magnitude against cDTW, while maintaining a retrieval accuracy very close to that of brute-force search.

Alexios Kotsifakos and Vassilis Athitsos
Department of Computer Science and Enginering
University of Texas at Arlington
E-mail: alexios.kotsifakos@mavs.uta.edu, athitsos@uta.edu

Isak Karlsson and Panagiotis Papapetrou
Department of Computer and Systems Sciences
Stockholm University
E-mail: isak-kar@dsv.su.se, panagiotis@dsv.su.se

Dimitrios Gunopulos
Department of Informatics and Telecommunications
National and Kapodistrian University of Athens
E-mail: dg@di.uoa.gr

1 Introduction

Finding the best matching subsequence to a query is a problem that has been attracting the attention of both database and data mining communities for the last few decades. The problem of *subsequence matching* is defined as follows: given a collection of sequences (database) and a query, identify the subsequence in the database that best matches the query. Achieving efficient subsequence matching is an important problem in domains where the target sequences are much longer than the queries, and where the best subsequence match for a query can start and end at any position in any sequence in the database.

A large number of distance or similarity measures based on Dynamic Programming (DP) [3] have been proposed for performing similarity search in many application domains and for several types of sequences, including time series, categorical sequences, and multimedia data. Nonetheless, there are still many application areas, such as music retrieval, where these methods are not directly applicable or have very poor retrieval accuracy [24], since in many cases several properties and characteristics of the specific domain are ignored. In this paper, we focus on time series subsequence matching and approach the problem from the music retrieval perspective: suppose you hear a song but you cannot recall its name; one solution is to hum a short part of the song and perform a search on a large music repository to find the song you are looking for or even songs with

similar melody. The main task of a *Query-By-Humming* (*QBH*) system is the following: given a hummed query song, search a music database for the K most similar songs. This directly maps to subsequence matching as the hummed query is typically a very small part of the target sequence. Let us now see how time series subsequence matching can be applied to QBH.

Every piece of music is a sequence of notes characterized by a *key*, that defines the standard pattern of allowed intervals that the sequence of notes should conform with, and a *tempo*, that regulates the speed of the piece. Each *note* consists of two parts: the *pitch* (frequency) and the *duration*. A *pitch interval* is the distance between two pitches. Another important term is *transposition*, the action of shifting a melody of a piece written in a specific key to another key. Finally, there is a discrimination between *monophonic* and *polyphonic* music, where for the latter it is possible for two or more notes to sound simultaneously. Here, we consider monophonic music, as in QBH we deal with melodies hummed by users.

Pitch and duration are two distinctive features for a music piece, and they should both be used for efficient music representation [43]. We could have two or more songs that share similar pitch values, but still have noticeably different melodies due to different individual pitch durations. In this work, we take into account both pitch and duration. Hence, melodies are defined as 2-dimensional time series of notes, where one dimension represents pitch and the other duration.

Robust and meaningful subsequence matching can be obtained in a potentially very noisy domain, like QBH, when several factors are considered. First, when humming a song, errors may occur due to instant key or tempo loss. Thus, the matching method should be *error-tolerant*, otherwise false negatives may dominate the retrieved results. In addition, the method should allow skipping a number of elements in both query and target sequences, so as to deal with temporary key or tempo loss. Nonetheless, the number of consecutive gaps in both query and target sequences should be bounded, in order to provide a setting that controls the expansion of the matched subsequences. Furthermore, to avoid producing very long matching subsequences, we should constrain the length of the matching subsequence, which can be set appropriately for the application domain. Also, to ensure that the matching subsequences will include as many elements as possible, the minimum number of matching elements may be lower bounded. However, such a bound may decrease the number of candidate matches. If we have prior knowledge about the singing skills of the person who hummed the queries, we can tighten or loosen this constraint for strong or weak hummers, respectively.

A subsequence matching framework, and a method called *SMBGT*, have been proposed [24] that address all above issues and have been shown to be highly suitable for noisy domains, such as QBH. A limitation of SMBGT is its time complexity, which is $O(|Q||X|)$, where Q is the query and X a database sequence. For application domains with many and large database sequences, such as QBH, the runtime is critical. Hence, a method for efficient similarity search under SMBGT is needed, that can achieve significant speedups against brute-force search with minor losses in accuracy.

In this paper, we propose a novel embedding-based filter-and-refine approach, which we call *ISMBGT*, shorthand for Indexed Subsequence Matching with Bounded Gaps and Tolerances. ISMBGT is designed to improve the efficiency of processing subsequence matching queries in time series databases under the SMBGT method. The key idea is that the subsequence matching problem can be partially converted to the much more manageable problem of nearest neighbor retrieval in a real-valued vector space. This conversion is achieved by defining an embedding function that maps each database sequence into a sequence of vectors, which we call database sequence embeddings. There is a one-to-one correspondence between each such vector and a position in the database sequence. The embedding function also maps each query sequence into a vector, which we call query embedding. The mapping is performed in such a way that if the query is very similar to a subsequence of the database, the new vector-based representation of the query is likely to be similar to the vector corresponding to the endpoint of that subsequence.

These vectors are computed by matching queries and database sequences with so-called *reference sequences*, which are a relatively small number of preselected sequences. The expensive operation of matching database and reference sequences is performed offline. At runtime, the query time series is mapped to a vector by matching the query with the reference sequences, which is typically orders of magnitude faster than matching the query with all database sequences. Then, promising candidates for the best subsequence match are identified by finding the nearest neighbors of the query vector among the database vectors. Applying sampling on the database vectors demonstrates that this process can be accomplished even faster. A refinement step is finally performed, where subsequences corresponding to the top vector-based matches are evaluated using the SMBGT method.

Contributions: We propose a novel embedding-based indexing approach, called ISMBGT, that works in a

filter-and-refine manner and is designed for speeding up similarity search under SMBGT. Furthermore, we provide an extensive experimental evaluation of ISMBGT using synthetic and hummed queries on a large music database. Our experimental results show that ISMBGT can achieve speedups of up to *an order of magnitude* against brute-force search under SMBGT, while maintaining a retrieval accuracy *very close to that of brute-force*.

We note that ISMBGT differs substantially from existing embedding-based indexing methods, in that (a) it does not require any training (which can be prohibitively costly for large databases) for creating the embeddings of the database sequences; instead, reference sequences are selected randomly from the database and cover all ranges of query lengths, (b) it is query-sensitive: the construction of the query vector is performed using a technique that optimizes the selection of the reference sequences based on the query length, and (c) it exploits the flexibility of SMBGT: it allows for a bounded number of gaps in both the query and target sequences, variable tolerance in the matching, a matching range in the alignment, and also constrains the minimum number of matching elements.

2 Related Work

Several DP methods exist for *whole sequence matching* including Dynamic Time Warping (DTW) [27] and variants (e.g., cDTW [40], EDR [7], ERP [6]) that are robust to misalignments and time warps. Other methods allow for gaps in the alignment, e.g., LCSS [30]. Zhu et al. [48] have applied DTW using the LB_Keogh lower-bound for music matching. These methods, however, are designed for whole sequence matching and not subsequence matching, which is our focus.

A method for improving the efficiency of subsequence matching under unconstrained DTW is described in Zhou and Wong [47], where it is assumed that the length of the optimal subsequence is known, and equal to the length of the query. Another method that handles shifting and scaling both temporal and amplitude dimensions based on time warping distance is called Spatial Assembling Distance (SpADe) [8]. The approach of Han et al. [14] is based on uniform segmentation and sliding windows, and requires the user to manually select the length of the segments. Pre-selecting the length of the segments is not a realistic assumption for an actual query-by-humming application, and therefore that method can efficiently handle only *near exact* matching. All aforementioned DP methods may fail to handle noise inherent in user performances in a QBH application.

Two subsequence matching methods have been proposed for melodic similarity [18,20] that implicitly account for local adjustments of tempo, though they are not transposition invariant. SPRING [41] is a DP-based method performing under DTW, that finds the subsequences of evolving numerical streams that are closest to a query. SPRING runs in time linear to both database and query sizes. Due to its construction, SPRING is sensitive to amplitude changes since it cannot support z-normalization; hence, we consider it inapplicable to our problem. The Edit distance [29] has been used for music retrieval with several variations [28]. LCSS-based approaches [4,5,43] can tolerate humming noise, though the fact that no bounds are imposed on the allowed gaps may result in a large number of false positives. Another approach has been proposed [9,19] that deals with *whole query matching*, employs a bounded number of gaps only in the target sequence, and does not account for duration. Some approaches embed transposition invariance as a cost function in the DP computation [28], though, with not attractive runtime. Kotsifakos et al. [24] proposed a subsequence matching framework and the SMBGT method, which was shown to be highly applicable to QBH. A survey of several DP-based methods has also demonstrated the competitiveness of SMBGT [25].

Furthermore, n-gram-based methods for music retrieval [10,43] fail to handle noisy queries efficiently, as they are designed for near exact matching. Also, probabilistic methods have been developed for speech recognition and music retrieval [35,38,45]. However, they are computationally expensive due to the required training, and creating models to represent all genres of music in a large database can be too challenging for real-world applications.

Several commercial products exist for music retrieval and QBH. Given a part of a song recording, Shazam (http://www.shazam.com) identifies the song. However, it only works for queries that are recordings of songs (with potential noise) having an exact match in the target database. The Soundhound application works for QBH (http://www.soundhound.com), as opposed to Shazam. However, it is not open-source and the technical details of its matching methods are unknown. Finally, "Hum-a-song" [26] is an open-source QBH application. Users are given a collection of similarity and distance measures and can freely choose one to perform QBH. Nonetheless, the purpose of this paper is not to provide a comparative evaluation of all QBH methods, but rather to propose methods for speeding up the computation of SMBGT.

An indexing structure for unconstrained DTW-based subsequence matching has been proposed in [36]. That

method, apart from being approximate, has the limitation that its complexity becomes similar to that of brute-force search as database sequences get longer. An alternative method [37] can handle such long database sequences by using monotonicity, but is only applicable to 1-dimensional time series. More importantly this method suffers from poor precision and recall. A related method that can be used for multidimensional time series is piece-wise approximation (PAA) [23], while a lower-bounding technique [21] and variants [42] have been proposed for time series matching. Finally, DTK [15] is a method for subsequence matching under cDTW. This approach, however, does not scale well as the query size increases. A similar approach is used to index time series for sequence and subsequence matching under scaling and DTW [13].

The indexing approach proposed in this paper is embedding-based. Several embedding methods exist for speeding up distance computations and nearest neighbor retrieval. Examples include Lipschitz embeddings [16], FastMap [11], MetricMap [46], SparseMap [17], and BoostMap [1,2]. Such embeddings can be used for speeding up full sequence matching [1,2,17]. Similar methods have been developed for subsequence matching in time series [33] and string [32] databases. Finally, an NN-search approach has been proposed [12] using dynamic vantage point trees. This approach is, however, limited to indexing objects in dimension spaces of up to 16-20 dimensions. As the dimensionality grows such methods become inappropriate for time series indexing. Overall, the above embedding-based methods are not directly applicable to SMBGT. In this paper, we propose an indexing method for SMBGT that achieves significant speedups against brute-force search.

We should note that embedding-based methods for non-metric distance measures, like the proposed SMBGT, are approximate and hence cannot always provide theoretical guarantees [33] of zero false negatives. This is in contrast to metric distance measures, for which embedding-based indexing methods with theoretical guarantees can be obtained [32]. Alternative approaches for exact indexing of specific time series distance measures have appeared in the literature. A method for exact indexing of EDR is proposed in [7], and is based on the "near triangle inequality" property of EDR. Such properties are not straightforward to prove for SMBGT. Nonetheless, our experimental evaluation demonstrates that the proposed indexing method can achieve substantial speedups, of over one order of magnitude compared to state-of-the-art methods, without sacrificing accuracy.

One way to represent notes is to encode only pitch [31]. Combining pitch with duration, though, improves accuracy in music retrieval. Two common ways to express pitch are: (a) *absolute pitch*, where the frequency of the note is used - in MIDI, the music format we are interested in, this value is an integer in $[1, 127]$ - and (b) *pitch interval*, the frequency difference between two adjacent notes. Three common ways to encode duration [34] are: (a) *Inter-Onset-Interval* (IOI), defined as the difference in time onsets of two adjacent notes, (b) *IOI Ratio* (IOIR), defined as the ratio of IOIs of two adjacent notes, and (c) *Log IOI Ratio* (LogIOIR), the logarithm of IOIR. In this paper we represent pitch using pitch intervals and duration using IOIR.

Last, we should note that in this paper, we study monophonic music rather than polyphonic [44], as it directly applies to QBH.

3 Problem Setting

In this section, we describe the representation that we use for the music pieces, we introduce the required notation and definitions, and we formally define the problem that we address in this paper.

3.1 Representing Music Pieces

In this paper, we use both pitch and duration to describe a music piece, and this results in a 2-dimensional time series representation. We consider the encoding scheme ⟨pitch interval, IOIR⟩. Thus, we deal with note transitions, saving much computational time as we do not have to check for possible transpositions of a melody, nor do we have to scale in time when comparing melodies. Apart from this, this representation leads to highest accuracies for several synthetic and hummed query sets compared to all other possible 2-dimensional encoding schemes [24].

3.2 Definitions

Let us now give a more general problem definition. Consider $X = (x_1, \ldots, x_{|X|})$ to be a multi-dimensional time series, where $|X|$ denotes the length of X. We use x_j^d to denote the d-th dimension of x_j, where $j = 1, \ldots, |X|$. In our case, where X represents a music piece, each $x_j = \langle x_j^1, x_j^2 \rangle \in X$ is a pair of real values, where x_j^1 and x_j^2 correspond to pitch and duration respectively, and are represented using the encoding of Section 3.1. A music database DB is a set of time series: $DB = \{X_1, \ldots, X_N\}$, where N is the number of music pieces in DB. A *subsequence* of X, denoted as

$X[ts : te] = \{x_{ts}, \ldots, x_{te}\}$, is an ordered set of elements from X appearing in the same order as in X. The first element of the subsequence is x_{ts} and the last is x_{te}. Note that $X[ts : te]$ is not necessarily continuous, i.e., gaps are allowed to occur by skipping elements of X. Let $Q = (q_1, \ldots, q_m)$ be another multi-dimensional time series, and consider the following definitions:

Definition 1 ***(Variable error-tolerant match)*** *Consider elements $q_i \in Q$ and $x_j \in X$. For each dimension d of x_j and q_i, we may define a function $\epsilon_d^f(i,j) = f(q_i^d, x_j^d)$. We say that q_i and x_j* match with variable ϵ-tolerance*, and denote it as $q_i \approx_\epsilon^f x_j$, if for each d a constraint $\mathcal{C}(q_i^d, x_j^d, \epsilon_d^f)$ is satisfied.*

Depending on the form of $\mathcal{C}$ we can define two types of variable tolerance, i.e., absolute and relative:

- ***variable absolute error tolerance:*** *values q_i^d and x_j^d may differ by at most ϵ_d^f:*

$$\mathcal{C}(q_i^d, x_j^d, \epsilon_d^f) = \{|x_j^d - q_i^d| \leq \epsilon_d^f\}. \quad (1)$$

- ***variable relative error tolerance:*** *value q_i^d may vary between a fraction and a multiple of x_j^d:*

$$\mathcal{C}(q_i^d, x_j^d, \epsilon_d^f) = \{x_j^d/(1+\epsilon_d^f) \leq q_i^d \leq x_j^d*(1+\epsilon_d^f)\}. \quad (2)$$

Note that if $\epsilon_d^f(i,j)$ is a constant function then the last two tolerances are called ***constant*** *absolute and relative error tolerances, respectively.*

It can be observed that the proposed variable tolerance framework is generic and can be used for any time series application domain. Next, we present an instantiation of the above definition for the 2-dimensional time series used in this paper.

Variable Tolerances - Instantiation: A reasonable definition for variable ϵ_1^f [24] (where $d = 1$ corresponds to the pitch dimension) is the following:

$$\epsilon_1^f(i,j) = f(q_i^1) = \lceil q_i^1 * t \rceil, \text{ with } t = 0.2,\ 0.25,\ 0.5. \quad (3)$$

Note that ϵ_1^f is a function of only q_i^1.

For pitch, in our evaluation we experimented with variable absolute error tolerance. For duration, which is represented by IOIR, we had to find a suitable form for $\mathcal{C}$. Hence, we asked people to hum several pieces of different kinds of music and we observed a tendency of making duration ratios smaller, even half of their actual values. This is reasonable, as users care more about singing melodies than being exact in tempo. Also, our definition of $\mathcal{C}$ should account for cases of queries at slower tempos. Thus, for IOIR, we define the following form of variable error tolerance without providing any function ϵ_2^f:

$$\mathcal{C}(q_i^2, x_j^2, -) = \{x_j^2 \leq 2 * q_i^2 \text{ and } x_j^2 - q_i^2 \geq -0.5\}. \quad (4)$$

Notice that this form of $\mathcal{C}$ is appropriate for variable error tolerance for the QBH application studied here.

Definition 2 ***(Common bounded-gapped subsequence)*** *Consider two subsequences $Q[ts_1{:}te_1]$ and $X[ts_2{:}te_2]$ of equal length. We use $\mathcal{G}_Q$ and $\mathcal{G}_X$ to denote the indices of those elements in Q and X, respectively, that are included in the corresponding subsequences. Let α and β be the number of consecutive elements that can be skipped in X and Q, respectively. If $q_{\pi_i} \approx_\epsilon^f x_{\gamma_i}$, $\forall \pi_i \in \mathcal{G}_Q$, $\forall \gamma_i \in \mathcal{G}_X$, $i = 1, \ldots, |\mathcal{G}_Q|$, and*

$$\pi_{i+1} - \pi_i - 1 \leq \beta,\ \gamma_{i+1} - \gamma_i - 1 \leq \alpha, \quad (5)$$

then, pair $\{Q[ts_1{:}te_1], X[ts_2{:}te_2]\}$ defines a common bounded-gapped subsequence *of Q and X. The longest such subsequence satisfying $te_2 - ts_2 \leq r$ is called* Longest Common Bounded-Gapped Subsequence *and denoted as $LCBGS(Q,X)$.*

Example. Consider a query $Q = (6, 3, 10, 5, 3, 2, 9)$ and a target sequence $X = (1, 1, 3, 4, 6, 9, 2, 3, 1)$. For simplicity, we assume that Q and X are 1-dimensional. Now, consider subsequence Q[2:6] with $\mathcal{G}_Q = \{2, 4, 5, 6\}$, which corresponds to sequence $(3, 5, 3, 2)$, and X[3:8] with $\mathcal{G}_X = \{3, 4, 7, 8\}$, which corresponds to $(3, 4, 2, 3)$. Also, we assume the following parameter settings: $\epsilon_1^f = 1$ (constant absolute tolerance) and $\epsilon_2^f = 0$ (since we only consider 1-dimensional time series), $\alpha = 2$, $\beta = 1$, and $r = 6$. Clearly, the two subsequences are of the same length, at most two ($\alpha = 2$) consecutive gaps occur in X (between the second and third elements in X[3:8]), and at most one ($\beta = 1$) consecutive gap occurs in Q (between the first and second elements in Q[2:6]). Range constraint $r = 6$ clearly holds for X[3:8], while all matching elements in the two subsequences differ by at most 1 ($\epsilon_1^f = 1$). Thus, pair $\{Q[2:6], X[3:8]\}$, is the $LCBGS(Q,X)$.

3.3 Problem Formulation

Problem: **(Top-K Subsequence Matching)** Given a database DB with N sequences of arbitrary lengths, a query sequence Q, and positive integers δ and r, find set

$\mathcal{S} = \{X_i[ts : te] | X_i \in DB\}$ with the K subsequences having the Longest $CBGS$, such that

$$|LCBGS(Q, X_i[ts : te])| \geq \delta. \quad (6)$$

In our setting, each database sequence contributes with only one subsequence $X_i[ts : te]$ to $\mathcal{S}$. The proposed method can be easily modified for supporting multiple subsequences to be returned within a single data sequence, given that they satisfy the searching criteria. This can be achieved by keeping a list of all matching subsequences for each channel. Finally, note the additional constraint $te - ts \leq r$, which is by definition included in LCBGS.

4 SMBGT: Subsequence Matching with Bounded Gaps and Tolerances

A method that is able to identify the LCBGS of a query Q and a sequence X is called $SMBGT$ [24]. Given K, SMBGT reports the K database sequences, where the K longest common bounded-gapped subsequences occur, which is essentially the solution to the Top-K Subsequence Matching Problem (Section 3.3).

SMBGT bounds the number of consecutive gaps allowed in *both* X and Q by two positive integers α and β, respectively. The intuition behind allowing gaps in both sequences is to deal with serious humming errors that are likely to occur due to temporary key/tempo loss or significant instant note loss (more than the acceptable tolerance). In addition, it allows for variable tolerance in the matching, the maximum length of the database subsequence with the longest common bounded-gapped subsequence is constrained by r, and the minimum number of matching elements by δ. Due to space limitations we skip the algorithmic details (the interested reader can refer to [24]). The appropriate tuning of the parameters depends on the application domain. An illustration of the methodology used by SMBGT is shown in Figure 1.

The space required is $O(|Q|)$ and the time complexity is $O(|Q||X|)$, which may be prohibitive for large databases. Moreover, since SMBGT is not metric, speeding up similarity search is more difficult. Thus, there is a need for an efficient indexing approach.

5 ISMBGT: Indexed SMBGT

In this section, we present ISMBGT, a novel approach for speeding up similarity search under SMBGT. ISMBGT exploits an embedding-based technique for indexing the database sequences. Given a query sequence, ISMBGT works in a filter-and-refine manner.

5.1 SMBGT Embeddings

The task at hand is to identify the top-K subsequence matches to a query Q in a database of sequences DB. The naive solution to this problem is to employ brute-force search by using the SMBGT method described in the previous section.

This paper proposes a more efficient method, which is based on defining a novel embedding function H, called *SMBGT embedding*. The key novelty of this embedding function is that it is specifically tailored for subsequence matching under SMBGT. Every element x_j of each database sequence X is mapped into an n-dimensional vector, called *SMBGT database sequence embedding*, and each query Q is also mapped into an n-dimensional vector, called *SMBGT query embedding*, where n is a positive integer. This mapping is performed via the use of a set of relatively short sequences, which we call *reference sequences*.

Let R be a reference sequence, Y be a target sequence (either database sequence or query), and $Y[i : j]$ be a subsequence of Y. We define $S_{|R|,j}(R, Y[i : j])$ to be the highest value of the last column of the alignment table α when we compute (in a dynamic programming manner) $LCBGS(R, Y[i : j])$ using SMBGT:

$$S_{|R|,j}(R, Y[i : j]) = max_{t=0,\ldots,|R|}\{\alpha_{t,|Y[i:j]|+1}\} \ . \quad (7)$$

We shall use R to define a 1-D SMBGT embedding.

Definition 3 ***(1-D SMBGT embedding)*** *Given a reference sequence R and a target sequence Y, function H^R is a 1-D SMBGT embedding that maps any subsequence $Y[i : j]$ of Y into a real number using R:*

$$H^R(Y, i, j) = S_{|R|,j}(R, Y[i : j]) \ . \quad (8)$$

Using the above definition, we can compute a 1-D SMBGT database sequence embedding as follows:

$$H^R(X, i, j) = S_{|R|,j}(R, X[i : j]) \ . \quad (9)$$

Regarding i, it is set to $j - c + 1$, where $c > 0$ is a constant determined by $|Q|$. Note that, in order to define i, it has to hold that $j \geq c$. However, if $|X| < c$ then we set $i = 1$ and $j = |X|$. The instantiation of c is given in Section 6.2.1.

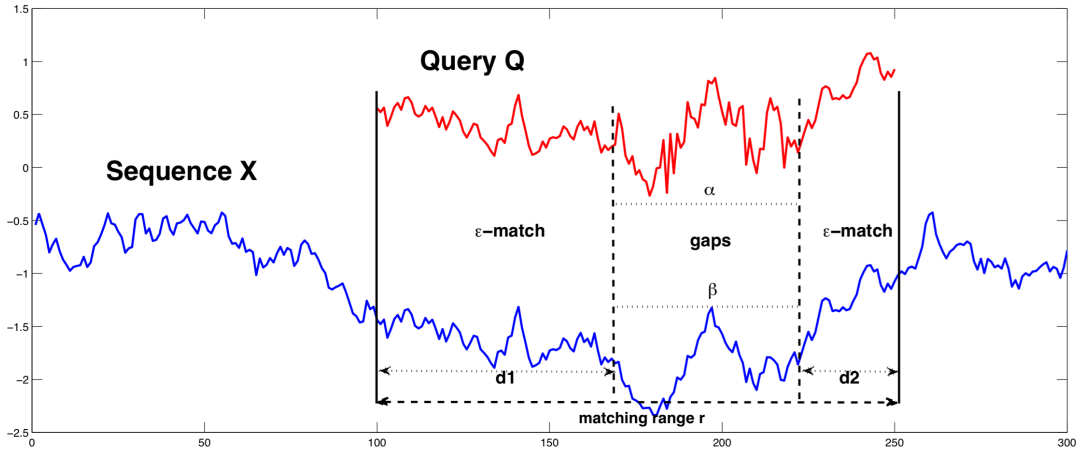

Fig. 1 An illustration of SMBGT given a query Q and a target sequence X, SMBGT finds the longest common bounded-gapped subsequence.

Similarly, a 1-D SMBGT query embedding is computed as follows:

$$H^R(Q, i, |Q|) = S_{|R|,|Q|}(R, Q[i : |Q|]) \ . \tag{10}$$

In this case we set $i = |Q| - c + 1$.

Consequently, for each reference sequence R, each database sequence is mapped to $|X| - c + 1$ values, if $|X| \geq c$, or one value if $|X| < c$. Note that for the case of the query, by definition, it holds that $|Q| \geq c$.

Naturally, instead of picking a single reference sequence R, we can pick multiple reference sequences to create a multi-dimensional embedding.

Definition 4 ***(n-D SMBGT embedding)*** *Given a set of n reference sequences $\mathcal{R} = \{R_1, \ldots, R_n\}$ and a target sequence Y, function $H^{\mathcal{R}}$ is an n-D SMBGT embedding that maps any subsequence $Y[i : j]$ of Y using $\mathcal{R}$ to the following vector:*

$$H^{\mathcal{R}}(Y, i, j) = \{S_{|R_1|,j}(R_1, Y[i : j]), \ldots, S_{|R_n|,j}(R_n, Y[i : j])\} \tag{11}$$

Hence, using Equations 9 and 10 we can compute the n-D SMBGT database sequence and query embedding vectors, respectively, as follows:

$$H^{\mathcal{R}}(X, i, j) = \{H^{R_1}(X, i, j), \ldots, H^{R_n}(X, i, j)) \ . \tag{12}$$

$$H^{\mathcal{R}}(Q, i, |Q|) = \{H^{R_1}(Q, i, |Q|), \ldots, H^{R_n}(Q, i, |Q|)\} \ . \tag{13}$$

5.2 Properties of SMBGT Embeddings

The construction of the SMBGT embeddings is performed in a way that if Q is similar to a subsequence of X starting at x_i and ending at x_j, and if R is a reference sequence, then $H^R(Q, i, |Q|)$ is likely to be similar to $H^R(X, i, j)$. Using the same argumentation as in Papapetrou et al. [33], we can see that: (i) if Q appears exactly as a subsequence $X[i : j]$ in X, it holds that $H^R(Q, i, |Q|) = H^R(X, i, j)$, given that the optimal alignment path computed by SMBGT when matching R with $X[i : j]$ does not start before position i, which is the position in X where the appearance of Q starts, and (ii) if $X[i : j]$ is a slightly perturbed version of Q in X, then $H^R(Q, i, |Q|) \approx H^R(X, i, j)$. In other words, the perturbed version of Q has been obtained, e.g., by adding some noise to each q_t, $t = 1, \ldots, |Q|$, of Q.

Thus, little tweaks in the values of Q should slightly affect the difference between the values of $H^R(Q, i, |Q|)$ and $H^R(X, i, j)$. This claim becomes even stronger when more reference sequences are used (see Papapetrou et al. [33]) which is the case in our approach. Unfortunately, due to the non-metric nature of SMBGT, there exists no approximation method that can make any strong theoretical guarantees in the presence of perturbations along the temporal axis. In order for the proposed embeddings to provide good retrieval accuracy, the following *statistical* property has to hold empirically:

Property: (**Embedding Similarity**) Let j_{opt} be the position in X where the optimal SMBGT subsequence match of Q in X ends. Then, given some random position $j \neq j_{opt}$, it should be very likely that

$H^{\mathcal{R}}(Q,i,|Q|)$ is closer to $H^{\mathcal{R}}(X,i,j_{opt})$ than to any other $H^{\mathcal{R}}(X,i,j)$.

This statistical property can be established by embedding optimization. It can be seen that the quality of the SMBGT embeddings depends on the selection of the reference sequence set $\mathcal{R}$. In Section 5.6, we describe a way of doing this, which is also one of the key differences of our approach against existing state-of-the-art embedding-based filter-and-refine techniques.

5.3 A Filter-and-refine Framework

Applying brute-force SMBGT on the database for a given query would be prohibitively computationally expensive for large databases. Our goal is to improve retrieval efficiency with very small loss in retrieval accuracy. Towards this objective, the design of ISMBGT enables a filter-and-refine nearest neighbor retrieval [16] using the SMBGT embeddings defined in the previous section. ISMBGT employs a filter step, where a small set of candidate database subsequences are selected, which are then passed to the refine step to perform the costly SMBGT computation. The substantial speedup is achieved by keeping the number of filtered subsequences quite small while allowing for very small loss in accuracy.

A pre-processing step is needed to create the SMBGT database embeddings. In particular, ISMBGT first performs a one-time offline step computation, where vector $H^{\mathcal{R}}(X,i,j)$ is computed for every allowable position j of each database sequence X. Computing the set of all n-D SMBGT database embeddings $H^{\mathcal{R}}(X,i,j)$, for $j \in [c,|X|]$ (with $|X| \geq c$) takes time $O(|X|\sum_{l=1}^{n}|R_l|)$. This process is repeated for all database sequences and the resulting set of embeddings corresponds to the embedding of the whole database. This set is stored and ISMBGT is then ready for receiving queries. As it can be seen, the complexity of this step depends purely on the length and number of reference sequences as well as the database length. Despite its complexity, this step is a one-time computation and can achieve substantial speedups in retrieval time without significant loss in accuracy (as shown in the experiments). At query time, qiven a query sequence Q, ISMBGT performs the following steps:

1. **Query embedding step:** The SMBGT query embedding $H^{\mathcal{R}}(Q,i,|Q|)$ is computed by applying the SMBGT method n times, one time for each of the n reference sequences. The total running time is $O(|Q|\sum_{l=1}^{n}|R_l|)$. This time is typically negligible compared to running SMBGT between Q and all sequences X in DB, which takes $O(|Q|\sum_{l=1}^{N}|X_l|)$ time, because we enforce $\sum_{l=1}^{n}|R_l| << \sum_{l=1}^{N}|X_l|$.
2. **Filter step:** $H^{\mathcal{R}}(Q,i,|Q|)$ is compared to $H^{\mathcal{R}}(X,i,j)$ of each database sequence, for $j = c,\ldots,|X|$, and some j's are chosen such that $H^{\mathcal{R}}(Q,i,|Q|)$ is very similar to $H^{\mathcal{R}}(X,i,j)$. Specifically, some pairs of database sequences X and positions j in those sequences (we denote them as (X,j)) are selected according to the distance between each $H^{\mathcal{R}}(X,i,j)$ and $H^{\mathcal{R}}(Q,i,|Q|)$. These are the *candidate endpoints* of the subsequences that best match with Q.
3. **Refine step:** For each such j, and for a matching range r, SMBGT is run between Q and $X[j-r+1:j]$ to find the best subsequence match. At the end of this step the subsequences with the K highest SMBGT scores, or else the K best matching subsequences to Q, will have been identified, thus solving the Top-K Subsequence Matching Problem. It is obvious that if we are able to choose a small number of promising subsequences $X[j-r+1:j]$, evaluating only those subsequences will be much faster than running SMBGT between Q and the whole database.

5.4 Filter Step

Original Scheme: We first select the best k reference sequences, i.e., the reference sequences that provide the k highest similarity scores in $H^{\mathcal{R}}(Q,i,|Q|)$. Then, for every (X,j) we compute the Euclidean distance between $H^{\mathcal{R}}(Q,i,|Q|)$ and $H^{\mathcal{R}}(X,i,j)$ using only the dimensions corresponding to the selected reference sequences. Finally, all database positions (X,j) are ranked in increasing order of the distance between $H^{\mathcal{R}}(X,i,j)$ and $H^{\mathcal{R}}(Q,i,|Q|)$, so that they can be further used at the refine step.

The time complexity is $O(k\sum_{l=1}^{N}|X_l|)$, which suggests that for large database sizes the filter step can still be expensive. Hence, we explore a sampling technique which, as we confirm in our experiments, can achieve significant speedups.

Speeding up the Filter Step: To reduce the computational cost of the filter step, ISMBGT performs sampling over the vector space of each SMBGT database sequence embedding. The intuition is that embeddings are constructed in a way that embeddings of nearby positions, such as $H^{\mathcal{R}}(X,i,j)$ and $H^{\mathcal{R}}(X,i,j+1)$, tend to be very similar.

We choose a sampling parameter s, and then sample uniformly one out of every s vectors $H^{\mathcal{R}}(X,i,j)$ in each X. In other words, we only store vectors:

$H^{\mathcal{R}}(X,i,1), H^{\mathcal{R}}(X,i,1+s), H^{\mathcal{R}}(X,i,1+2s), \ldots$.

Given $H^{\mathcal{R}}(Q,i,|Q|)$, we compute its distance with the vectors that we have sampled, using only the k dimensions that correspond to the reference sequences that provide the best k similarity scores in $H^{\mathcal{R}}(Q,i,|Q|)$ (similar to the original scheme). If for a database position (X,j) its vector $H^{\mathcal{R}}(X,i,j)$ was not sampled, then we assign to that position the distance between $H^{\mathcal{R}}(Q,i,|Q|)$ and the vector that was actually sampled among $\{H^{\mathcal{R}}(X,i,j-\lfloor s/2 \rfloor), \ldots,$
$H^{\mathcal{R}}(X,i,j+\lfloor s/2 \rfloor)\}$. The time complexity of this step is $O(k \sum_{l=1}^{N} \lceil \frac{|X_l|}{s} \rceil)$. In our experiments, we observed that sampling achieves significant runtime improvements compared to brute-force search, without essential loss in accuracy.

5.5 Refine Step

As mentioned above, the filter step ranks all database positions (X,j) in increasing order of the distance, or estimated distance when we use approximations such as sampling, between $H^{\mathcal{R}}(X,i,j)$ and $H^{\mathcal{R}}(Q,i,|Q|)$. The refine step then evaluates a percentage *perc* of the total number of positions (which are ranked), where *perc* is a system parameter that provides a trade-off between retrieval accuracy and efficiency.

Since candidate positions (X,j) actually represent candidate *endpoints* of a subsequence match, we can evaluate each such candidate endpoint by performing the SMBGT method from a startpoint determined by r up to this endpoint. Specifically, if j_{end} is the endpoint of a potential match, we run the SMBGT method between Q and $X[j_{\text{end}} - r + 1 : j_{\text{end}}]$. In other words, SMBGT is used to find the best matching score and the corresponding subsequence for that candidate endpoint. We should mention that if we do not put any constraint on the value of the matching range parameter r, or else set r to infinity, SMBGT will be evaluated from the beginning of the database sequence X. However, subsequences of X that are much longer than Q are very unlikely to be good matches for Q, since there will be many unmatched elements creating large gaps when aligning the two sequences with SMBGT.

After all similarity scores between Q and candidate positions (X,j) have been computed, we sort them in decreasing order and check if the correct/targeted sequence is included in the returned results. If so, then the rank of Q is the position of the correct sequence. Otherwise, the rank of the query is the rank of the targeted sequence when we apply ISMBGT to all endpoints, and not just the *perc* of them. The time complexity of this step is $O(r\ |Q|\ perc\ \sum_{l=1}^{N} |X_l|)$.

5.6 Query-optimized reference sequences

It can be easily seen that the construction of the SMBGT query embedding is highly dependent on the length difference between the query and the reference sequences. If the length difference is high (the reference sequences are either much smaller or much longer than the query), then this will produce a low SMBGT similarity score. In order to guarantee that the Embedding Similarity Property holds, we should always choose reference sequences that are shorter than the queries but not too much shorter either. This has also been argued in Papapetrou et al. [33].

Hence, we employ a novel technique for building the SMBGT query embeddings. We first introduce a lower and an upper bound of the possible query lengths that our filter-and-refine framework may accept, denoted as L_Q and U_Q, respectively. Next, we split the query lengths into d' intervals, which results in a set of d' *buckets* $\mathcal{B} = \{b_1, \ldots, b_{d'}\}$. Each backet b_i corresponds to interval $[uv^{i-1}, uv^i)$, for $i = 1, \ldots, d'$, where $u = L_Q$ and $uv^{d'} = U_Q$. The resulting set of buckets is the following:

$$\mathcal{B} = \{[u, uv), [uv, uv^2), [uv^2, uv^3), \ldots [uv^{d'-1}, uv^{d'})\} \tag{14}$$

Then, we assign a set of reference sequences to each bucket b_i that satisfies the corresponding length requirement of b_i. Specifically, for each bucket b_i, the number of reference sequences is set to q, and their length is defined as a percentage p of the lower bound of the bucket, i.e., puv^{i-1}. This process is performed offline as well as the construction of the embedding of the whole database.

At query time, given Q, we identify the bucket $b_i = [uv^{i-1}, uv^i)$, such that $uv^{i-1} \leq |Q| < uv^i$. The SMBGT query embedding is then created using the set of q reference sequences that correspond to bucket b_i.

6 Experiments

Kotsifakos et al. [24] have demonstrated the superiority of SMBGT in terms of accuracy over several DP-based methods and an HMM-based method on QBH. In that study, both synthetic and hummed queries have been tested on a large variety of music representations and

error tolerance types. Here, we demonstrate the usefulness of ISMBGT when compared to the brute-force SMBGT and cDTW in terms of retrieval accuracy and runtime.

6.1 Experimental Setup

Database: Our music database consists of $40,891$ 2-dimensional time series sequences of variable length and a total of $13,455,603$ 2-dimensional tuples. The database was created based on a set of $5,643$ MIDI files, freely available on the web. These MIDI files cover various music genres, such as Blues, Rock, Pop, Classical, Jazz, and also themes from movies and TV series. Each MIDI file comprises at most 16 channels. For each MIDI file and channel, we extracted the highest pitch at every time click (*all-channels extraction* [43]). Then, we converted tuples ⟨pitch, click⟩ to ⟨pitch interval, IOIR⟩, resulting in $|DB| = 40,891$ sequences and a total of $13,455,603$ tuples. This pre-processing procedure was done offline and only once, guaranteeing that there is no possibility of missing a melody existing in any channel.
Synthetic Queries: Six synthetic query sets (100 queries per set) of lengths in $[13, 137]$ were generated: Q_0, $Q_{.10}$, $Q_{.20}$, $Q_{.30}$, $Q_{.40}$, and $Q_{.50}$. Q_0 contained 100 exact segments of DB, while $Q_{.10} - Q_{.50}$ were generated by adding noise to each query in Q_0. For all queries of Q_0 we randomly modified 10, 20, 30, 40, and 50% of their corresponding time series in both dimensions; pitch interval by $\pm z \in [3, 8]$ (simulates error performed when singing by memory and intrinsic noise that may be added by audio processing tools), IOIR by $\pm z \in [2, 4]$ (simulates reasonable variations of duration and is outside the bounds of Equation 4, avoiding bias in favor of SMBGT). In all query sets we allowed at most 3 consecutive elements to be modified (no insertions or deletions); in QBH we do not expect to have too many consecutive matching errors.
Hummed Queries: We also experimented with 100 hummed queries of lengths in $[14, 76]$ [24]. Users were asked to sing to a microphone and avoid lyrics. Melodies were converted to MIDI using *Akoff* [1], a well-known tool for evaluating QBH systems [48]. All-channels extraction was applied to the queries to obtain the representation of DB. The query set covered several genres, such as Classical, Blues, Jazz, Rock 'n' Roll, Rock, and Country [24].

An astute reader may notice that the number of people who helped in the construction of this set is small. However, creating such a set is not trivial, since selecting the final set of queries involved significant manual processing. Users made mistakes and the recording introduced noise (in both pitch and duration) to each song. Thus, users had to hum each song several times before selecting the version of their hummed song with the least noise (the one whose melody sounded close to the target according to them). The database and queries are available online [2].
Evaluation: We evaluated the performance of ISMBGT and brute-force SMBGT for the five noisy synthetic query sets and the hummed query set in terms of *recall*, *runtime*, and *efficiency*. We also evaluated the performance of cDTW on the same sets in terms of recall and runtime. For cDTW, we have used an optimized version based on the UCR_Suite [39]. Since the original code was designed for 1-dimensional time series, we have extended the code for 2-dimensional time series. The code is available online[3].

Recall is the number of queries for which the correct answer is in the top-K results, while efficiency is a particularly useful measure for evaluating ISMBGT that influences the retrieval runtime. Efficiency is the ratio of the number of database elements evaluated during the refine step by SMBGT to the length of the database. Experiments were run on an AMD Opteron 8220 SE processor at 2.8GHz, and implemented in C++.

6.2 Evaluation of ISMBGT

6.2.1 Reference Sequences

We follow the process described in Section 5.6 to select reference sequences for the construction of the SMBGT embeddings. We first create the set of buckets $\mathcal{B}$. Since query lengths are in $[13, 137]$, we set $L_Q = 13$ and $U_Q = 138$. Also, we set $d' = 5$, which means that we consider 5 query length intervals. Thus, $v \approx 1.60$. Next, for each bucket we set $q = 1,000$ and $p = 0.8$. Thus, we randomly selected $1,000$ sequences from DB with their lengths being 80% of the lower bound of the bucket. As a result, $n = 1,000$, i.e., we constructed 1,000-dimensional SMBGT embeddings. We should note that the construction of the SMBGT database sequences embeddings is done offline. The value of c needed when creating the SMBGT embeddings is set to the lower bound of the bucket that the length of the query belongs to. Hence, c is 13, 21, 33, 53, and 86 for each of the 5 buckets. We will use the terms (query length) intervals and buckets interchangeably. The lower and upper bounds of the buckets, the lengths of the reference sequences, and the c values are shown in Table 1.

[1] `http://www.akoff.com/music-composer.html`

[2] `http://vlm1.uta.edu/~akotsif/ismbgt/`

[3] https://github.com/isakkarlsson/ucr-suite-2d

Table 1 Buckets statistics.

Interval Notation	Interval	c	$\lvert R\rvert$	Hummed Queries	Synthetic Queries
b_1	$[13, 21)$	13	10	15	25
b_2	$[21, 33)$	21	17	30	90
b_3	$[33, 53)$	33	26	39	180
b_4	$[53, 86)$	53	42	16	135
b_5	$[86, 138)$	86	69	-	70

6.2.2 Parameters and Implementation details

In order to be fair when comparing brute-force SMBGT (for the remainder of this section, brute-force SMBGT will also be denoted as BF) with ISMBGT, we set the parameters as shown by Kotsifakos et al. [24], since these values achieved the best accuracy for BF. For the synthetic queries, the number of gaps that can be skipped in both query and target sequences is 3 ($\alpha = \beta = 3$), the matching range r is $|Q|$, and no matching tolerance is imposed so as to enforce the intact elements of the queries to be matched with the corresponding ones from the targeted subsequences. The minimum number of elements δ that have to match is $0.1*|Q|$. For the hummed queries, variable absolute tolerance with $t = 0.2$ is applied for pitch (Equations 1 and 3) and variable error tolerance (Equation 4 - IOIR) for duration, $\alpha = 5$, $\beta = 6$, $r = 1.2*|Q|$, and $\delta = 0.1*|Q|$ so as to have at least some elements of Q to match with the subsequences returned in the top-K results. Regarding cDTW, we employed the sliding window technique and, similarly to SMBGT, parameter r was set to $|Q|$ for the synthetic queries and to $r = 1.2 * |Q|$ for the hummed queries. For the Sakoe-Chiba band parameter, it was set to 4 for the synthetic queries to ensure that the method can successfully capture the maximum number of gaps in both query and target sequences, and to 10 for the hummed queries, which was the value achieving the best overall tradeoff between accuracy and runtime.

With regard to the percentage of endpoints evaluated at the refine step of ISMBGT, after experimenting with $perc = 0.1 - 0.9\%$ with step 0.2 and $perc = 1 - 5\%$ with step 2, we set it to 1%. This choice accounts for the trade-off between accuracy and runtime. Lower values of *perc* did not provide satisfying recall, although speedup improvements were higher than for $perc = 1\%$, while greater values significantly decreased speedup. $perc = 1\%$ makes ISMBGT at least 2 times faster than BF (Section 6.2.4), which is important in a real QBH scenario, especially if we consider that it leads to very high recall values.

At the refine step, we keep for each sequence X corresponding to a channel an array of size $|X|$ initialized with zeros. Then, for each candidate endpoint j_{end} that belongs to the *perc* of the sorted endpoints, we put value one to all positions of the array starting from j_{end} and going backwards up to $j_{\text{end}} - r + 1$. After that operation, we trace each sequence that has ones in its array and perform SMBGT between Q and the segments filled in with ones. If there are more than one segments for X we keep the best similarity score obtained after applying SMBGT to all of them as the score for X. The scores of the sequences are finally sorted, and it is checked whether any channel of the correct/targeted song is included in these sequences. If there exists a channel of the correct song, then the rank of the query is the rank of the channel of the correct song with the highest score (worst in case of ties). Otherwise, the rank of the query is the (worst in case of ties) rank of the closest channel of the song (the one with the highest score) when we apply ISMBGT to all endpoints, and not just the *perc* of them.

6.2.3 Accuracy

We first evaluated BF and ISMBGT in terms of accuracy on the five noisy synthetic query sets and the hummed query set. For completeness, we mention that the approach followed to find the recall for BF is the same as that of ISMBGT; the correct channel of the targeted song is the channel with the highest similarity score among all channels of that song.

In Figures 2-6 we present the performance of BF and ISMBGT in terms of recall vs. the top-$5, 10, 20, 50, 100$ returned results for the five intervals of the synthetic queries, respectively. Similar results are presented for the hummed queries in Figures 7-9. The figures illustrate what recall can be achieved when varying the number of the k best reference sequences used in the filter step and the sampling parameter s (for ISMBGT). For each interval, we present the recall vs. top-K for (a) different values of k when there is no sampling during the filter step ($s = 1$), (b) different values of k when $s = 3$, and (c) different values of s, for the value(s) of k that seemed more promising when we applied sampling with $s = 3$. In all figures, for ease of readability, ISBMGT is denoted as EMB. It is clear that high recall values are desirable for small K, while s and k are as high and small, respectively, as possible. Next, we present for each interval the combination of k and s values that attained the highest recall curve while still achieving significant speedup compared to BF.

Starting with the synthetic queries and b_1 (Figure 2), BF achieves 64% recall for the top-K=5 results (16 correct results out of 25 queries) and 84% for $K = 10, 20, 50$ (21 correct results), while ISMBGT for $k = 3$ and $s = 5$ achieves 72% recall for the top-$5, 10, 20, 50$ results. This shows that by even retrieving as few as the

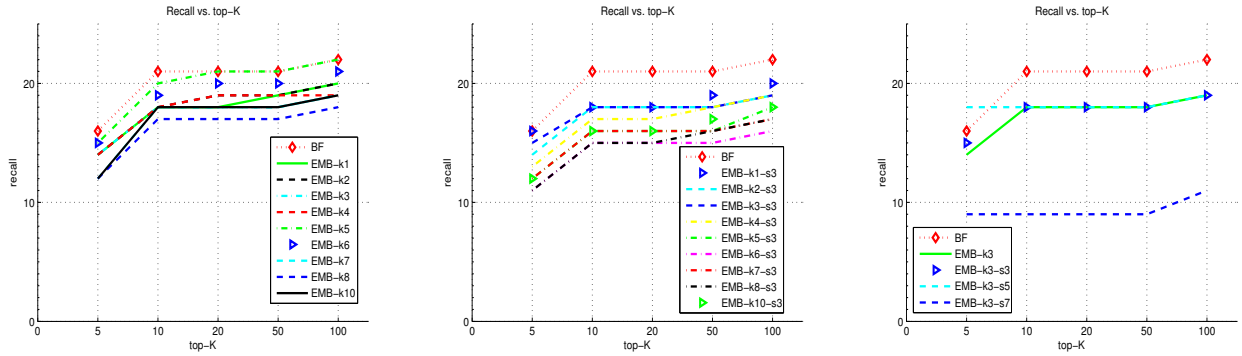

Fig. 2 Recall for the synthetic queries of bucket b_1. Recall is shown for BF and ISMBGT for $perc = 1\%$, the best k reference sequences, and sampling parameter s: (left) $k = 1-8, 10$ with no sampling ($s = 1$); (mid) $k = 1-8, 10$ with $s = 3$; (right) $k = 3$ with $s = 1, 3, 5, 7$.

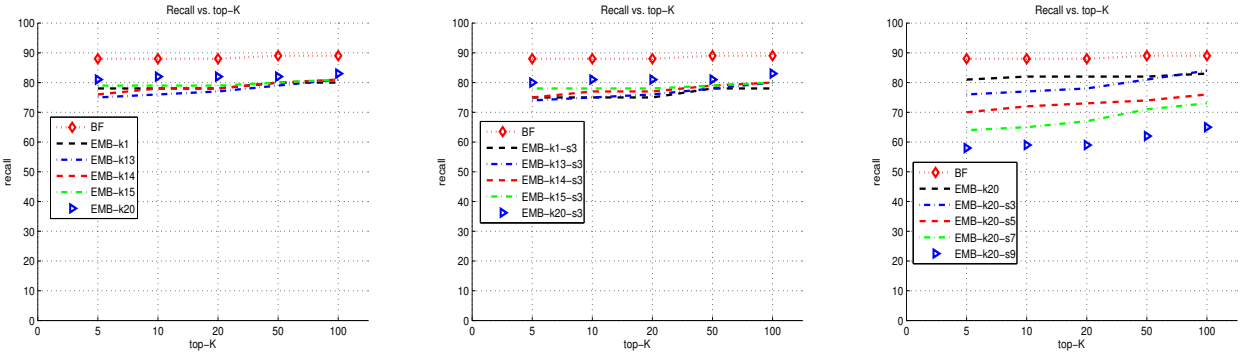

Fig. 3 Recall for the synthetic queries of bucket b_2. Recall is shown for BF and ISMBGT for $perc = 1\%$, the best k reference sequences, and sampling parameter s: (left) $k = 1, 13-15, 20$ with no sampling ($s = 1$); (mid) $k = 1, 13-15, 20$ with $s = 3$; (right) $k = 20$ with $s = 1, 3, 5, 7, 9$.

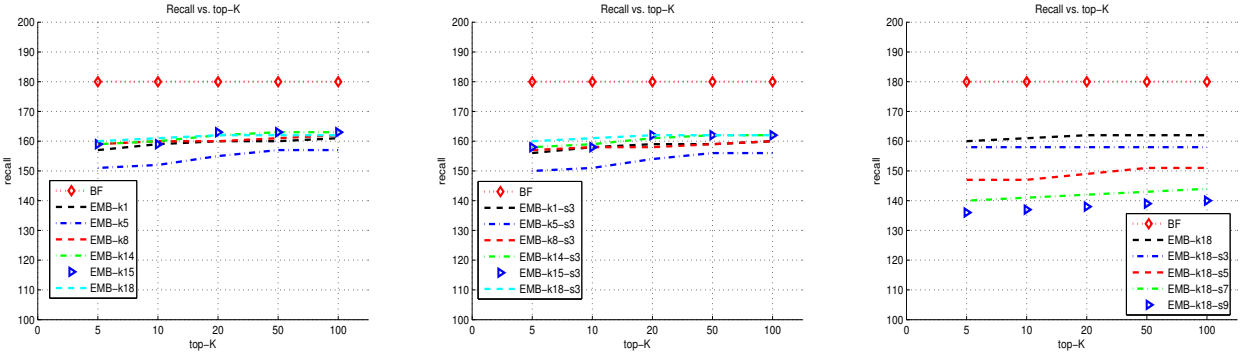

Fig. 4 Recall for the synthetic queries of bucket b_3. Recall is shown for BF and ISMBGT for $perc = 1\%$, the best k reference sequences, and sampling parameter s: (left) $k = 1, 5, 8, 14, 15, 18$ with no sampling ($s = 1$); (mid) $k = 1, 5, 8, 14, 15, 18$ with $s = 3$; (right) $k = 18$ with $s = 1, 3, 5, 7, 9$.

top-5 results the indexing method achieves higher recall than the BF, which is ideal. For b_2 (Figure 3) BF has a recall of 97.77% (88 correct results out of 90 queries) for $K = 5, 10$ and 98.88% for $K = 50$, while ISMBGT gives 84.44% when $k = 20$, $s = 3$ and $K = 5$ (76 correct results), and 85.55%, 86.66%, and 90% for $K = 10, 20, 50$, respectively. This difference of approximately 10% is perfectly acceptable, especially if we consider the runtime speedup and efficiency evaluation measure when compared to BF, as shown in the next sections and Tables 5 and 7. Similar conclusions hold for intervals b_3, b_4 and b_5 (Figures 4-6). More specifically, BF has 100%

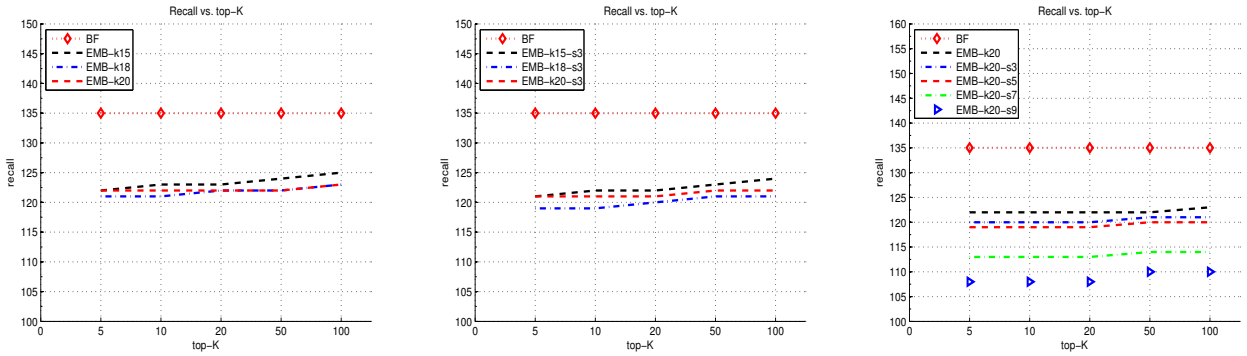

Fig. 5 Recall for the synthetic queries of bucket b_4. Recall is shown for BF and ISMBGT for $perc = 1\%$, the best k reference sequences, and sampling parameter s: (left) $k = 15, 18, 20$ with no sampling ($s = 1$); (mid) $k = 15, 18, 20$ with $s = 3$; (right) $k = 20$ with $s = 1, 3, 5, 7, 9$.

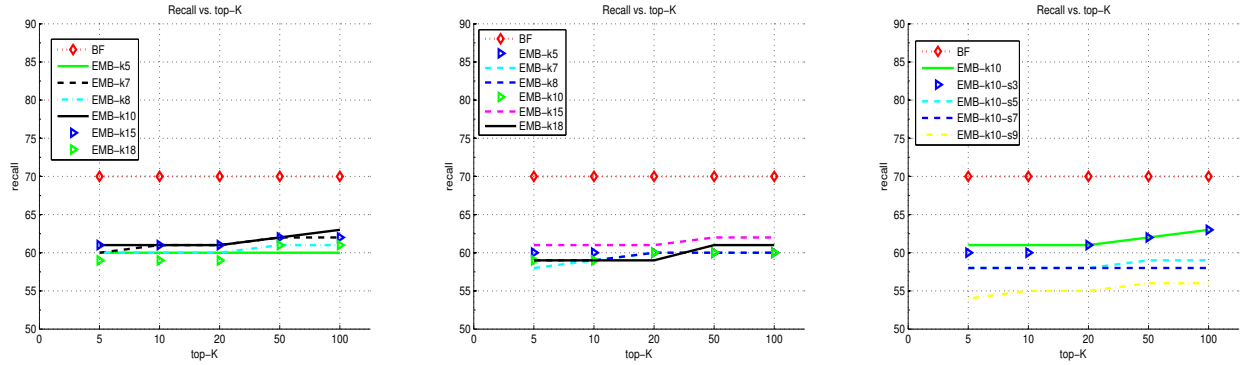

Fig. 6 Recall for the synthetic queries of bucket b_5. Recall is shown for BF and ISMBGT for $perc = 1\%$, the best k reference sequences, and sampling parameter s: (left) $k = 5, 7, 8, 10, 15, 18$ with no sampling ($s = 1$); (mid) $k = 5, 7, 8, 10, 15, 18$ with $s = 3$; (right) $k = 10$ with $s = 1, 3, 5, 7, 9$.

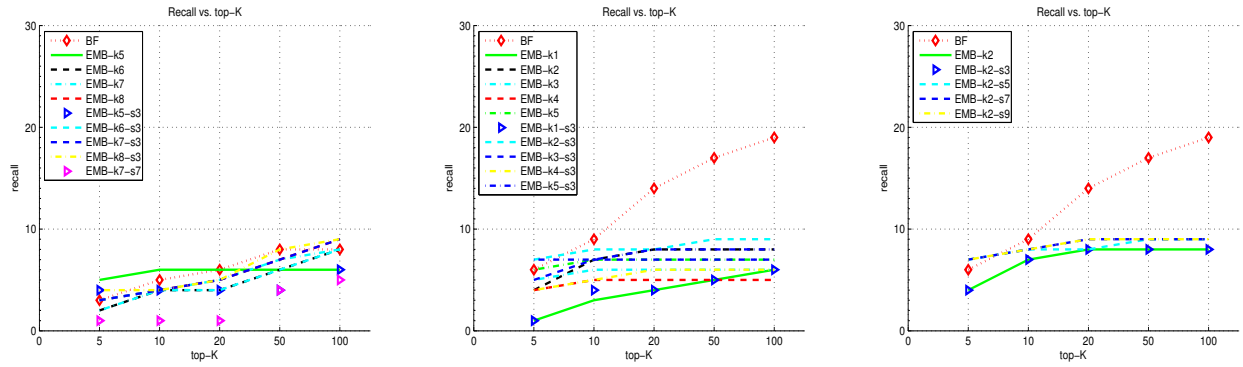

Fig. 7 Recall for the hummed queries of buckets b_1 and b_2. Recall is shown for BF and ISMBGT for $perc = 1\%$, the best k reference sequences, and sampling parameter s: (left) bucket b_1: $k = 5 - 8$ with $s = 1, 3$ and $k = 7$ with $s = 7$; (mid) bucket b_2: $k = 1 - 5$ with $s = 1, 3$; (2.3) bucket b_2: $k = 2$ with $s = 1, 3, 5, 7, 9$.

recall for all of these intervals when $K = 5$. Regarding ISMBGT, for b_3 with $k = 18$ and $s = 3$ it achieves 87.77% recall for $K = 5$ (158 correct results out of 180 queries). For b_4 with $k = 20$, $s = 5$, and $K = 5$ its recall is 88.15% (119 correct results out of 135 queries) while for 50 results it slightly increases to 88.88%. With respect to the bucket of largest query lengths, i.e., b_5, ISMBGT with $k = 10$, $s = 3$, and $K = 5$ has 85.71% recall and 87.14% for $K = 20$ (61 correct answers out of 70 queries).

With regard to the hummed queries, we observe the following accuracies. The query songs of b_1 are very

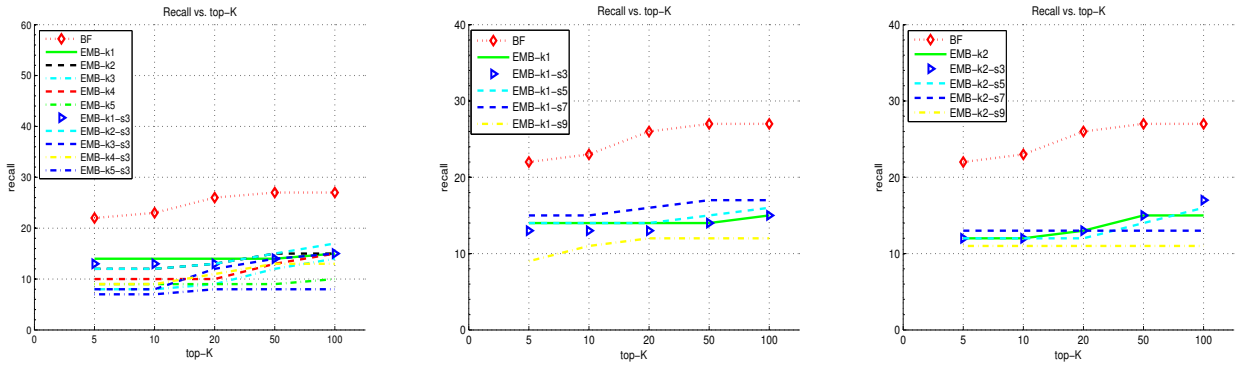

Fig. 8 Recall for the hummed queries of bucket b_3. Recall is shown for BF and ISMBGT for $perc = 1\%$, the best k reference sequences, and sampling parameter s: (left) $k = 1 - 5$ with $s = 1, 3$; (mid) $k = 1$ with $s = 1, 3, 5, 7, 9$; (right) $k = 2$ with $s = 1, 3, 5, 7, 9$.

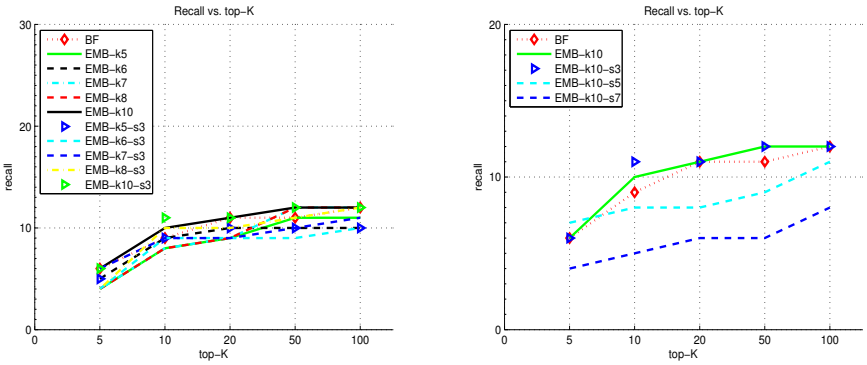

Fig. 9 Recall for the hummed queries of bucket b_4. Recall is shown for BF and ISMBGT for $perc = 1\%$, the best k reference sequences, and sampling parameter s: (left) $k = 5 - 8, 10$ with $s = 1, 3$; (right) $k = 10$ with $s = 1, 3, 5, 7$.

hard to identify even by BF (Figure 7(left)), since for $K = 5$ there are only 3 correct answers, while for $K = 10, 20$, and 50 there are 5, 6, and 8 correct answers, respectively. As happens with BF, ISMBGT with $k = 7$, $s = 3$ and $K = 5$ has 3 correct results, while for $K = 10, 20$, and 50 it has 4, 5, and 7, respectively. Thus, we can clearly see that although the query lengths are very small for b_1, ISMBGT is almost identical with BF, and that the number of reference sequences needed in the filter step is relatively low ($k = 7$). Referring to s, it cannot be greater than 3, since the small query lengths do not allow to skip more endpoints during the filter step and still attain good retrieval accuracy. Similar to the first interval, BF for b_2 (Figure 7(mid-right)) attains 6 out of 30 queries to have their correct answer in the top-5 results, while ISMBGT with $k = 2$ and $s = 7$ has 7 correct results (for $K = 5$), even higher than BF. For $K = 10$ and 20 there are 9 and 14 correct results for BF, while for ISMBGT we get 8 and 9. The reason for not getting higher recall is that only 2 reference sequences are used, and s is very high for a real QBH scenario. Queries of b_3 (Figure 8) are again very hard to identify. BF has 22 queries (out of 39) with their targeted song in the top-5 results, while ISMBGT has 15 with k being only 1 and $s = 7$. This combination of values is extreme for our proposed indexing approach and QBH, and thus different parameters can be used to increase recall (but decrease speedup). For b_4 (Figure 9) BF decides correctly for 6 (out of 16) queries when $K = 5$, while ISMBGT for more, i.e., 7, with $k = 10$ and $s = 5$. When $K = 10$ and 20 BF has 9 and 11 correct results, and ISMBGT 8 for both values of K. In Tables 2 and 6.2.3 we provide a summary of the accuracy results obtained by BF and ISMBGT, respectively, for the synthetic and hummed queries for all buckets.

In Table 4 we also provide the accuracy of cDTW. Our observations are that for the synthetic queries and small query lengths, i.e., belonging to buckets b_1 and b_2, ISMBGT yields better accuracies, while for larger query lengths, i.e., $b_3 - b_5$, cDTW is better. Referring to the hummed queries, for b_1 both methods provide almost the same accuracies, for b_2 and b_3 ISMBGT is better for $K = 5, 10, 20$ and worse for higher K values (50 and 100), while for b_4 ISMBGT is consistently better (for

all K values cDTW has 0 correct queries in the result set). These results indicate that, in general, for real queries the proposed ISMBGT method provides better accuracies. At the end of Section 6.2.4 we also discuss the runtimes of cDTW, showing that it is significantly slower than our method.

6.2.4 Time

In Table 5 the runtimes of BF and ISMBGT are shown for the synthetic and hummed queries (suffix '-S' and '-H') for each bucket. The runtimes of ISMBGT correspond to the best recall that was achieved (Section 6.2.3). The *ratio* of brute-force SMBGT runtime to that of ISMBGT per bucket is also provided. Ratio is the indicator of how well ISMBGT performs in terms of speedup over brute-force. In addition, although no sampling gives better accuracies than applying it, in Table 5 we show the runtimes (and ratios) for the values of s and k that yield the highest recall when $s > 1$. This is because we want to maximize recall while gaining as much as possible in runtime.

It is clear that the ratio is much greater for the hummed queries than the synthetic ones for every bucket. Specifically, the ratios for the synthetic queries for b_1-b_5 are 1.99, 2.25, 3.2, 4.28, and 5.85, while for the hummed 2.72, 7.82, 4.47, and 8.32 (no queries for b_5). The main reason for gaining more in terms of runtime for the hummed queries is that the BF-H runtimes are on average 2 times higher than those of BF-S. This is due to the computation scheme of SMBGT, which is more complex for the hummed queries than for the synthetic ones. There is no tolerance for the synthetic queries, while for the hummed variable absolute tolerance with $t = 0.2$ is used, which is a more sophisticated tolerance type. Also, ISMBGT runtimes do not differ much between 'S' and 'H'. For no interval is ISMBGT-H runtime greater than 1.67 times the ISMBGT-S runtime, while ISMBGT-H runtime for b_2 is even smaller than ISMBGT-S runtime by 1.63 times. The reason for this decrease is that for ISMBGT-S $k = 20$ and $s = 3$, while for ISMBGT-H k is an order of magnitude smaller and s is more than 2 times greater, i.e., $k = 2$ and $s = 7$. This combination leads to computing distances between much smaller and fewer vectors during the filter step.

Another important observation is that the ratio values increase as we move from b_1 to b_5 for both synthetic and hummed queries. The main reason for this is that the runtimes of BF-S and BF-H increase as well, but more than the runtimes of ISMBGT-S and ISMBGT-H. The complexity of ISMBGT depends on the length of the queries and the length of the segments of the sequences that are evaluated during the refine step. These segments get bigger as we move towards b_5, because they are created based on r, which in turn is a factor of $|Q|$. We note that the ratio for the hummed queries for b_2 is 7.82, which is greater than the 4.47 ratio for b_3. Although $s = 7$ for both buckets and $k = 2$ for b_2 while it is $k = 1$ for b_3, the difference in the query lengths between those intervals makes ISMBGT-H yielding much lower total runtime for b_2 (3.68 seconds) than for b_3 (8.71 seconds).

The average runtimes for each step of ISMBGT (embedding, filter, refine) per interval for the synthetic and hummed queries are given in Table 6. We observe that the embedding step times increase per interval for both synthetic and hummed queries. This is reasonable because the time for this step depends on the complexity of SMBGT that is influenced by $|Q|$ and $|R|$, which are both increased. The maximum value for synthetic and hummed queries appears for b_5 and b_4, and is only 0.15 and 0.21 seconds, respectively.

Regarding the filter step times, they depend on k and s. For synthetic queries and b_1 the runtime is only 0.58 seconds, because k is only 3 and s is relatively high, i.e., $s = 5$. For b_2, b_3, and b_4 the filter times are much greater, i.e., 2.76, 2.55, and 2.13 seconds, since k is 18, 20, and 18, respectively. For b_4 the filter time is smaller than those of b_2 and b_3 because of using larger s (5 instead of 3). This indicates that even though $|Q|$ and $|R|$ are higher for b_4, sampling plays a significant role in the speedup of ISMBGT. Runtime for b_5 is 1.63 seconds, mainly because of $k = 10$. For the hummed queries, the smallest runtimes are for b_2 and b_3 (0.36 and 0.3 seconds), as k is only 2 and 1, respectively. Since $s = 7$ for both buckets, and they only differ in k, time is slightly greater for b_2. Also, the higher value of k, which is 7, for b_1 leads to just over 2 seconds. What is more, although k is greater for b_4 than b_1, b_4 has smaller time since $s = 5$ (for b_1 $s = 3$). The maximum value for the synthetic queries is 2.76 (bucket b_2) and for the hummed 2.04 seconds (bucket b_1), confirming the importance of having a fast filter step.

For the refine step of ISMBGT and the synthetic queries the time increases when we examine intervals of greater lengths. This is because the runtime of this step depends on $|Q|$ and the lengths of the segments of sequences to which SMBGT is applied, while the number of these sequences does not deviate much. For the hummed queries, there is no trend in runtimes, verifying the dependence of this step on the number of sequences to which SMBGT is applied. The maximum time for the refine step and the synthetic queries is 5.93 (bucket b_5), while for the hummed queries it is 8.36 seconds (bucket b_3), showing that refine essentially determines the total runtime of ISMBGT.

Table 2 Accuracy (number of correct queries) of brute-force SMBGT for synthetic and hummed queries.

	Accuracy of brute-force SMBGT								
	Synthetic					**Hummed**			
top-K	b_1	b_2	b_3	b_4	b_5	b_1	b_2	b_3	b_4
$K = 5$	16	88	180	135	70	3	6	22	6
$K = 10$	21	88	180	135	70	5	9	23	9
$K = 20$	21	88	180	135	70	6	14	26	11
$K = 50$	21	89	180	135	70	8	17	27	11
$K = 100$	22	89	180	135	70	8	19	27	12

Table 3 Accuracy (number of correct queries) of ISMBGT for synthetic and hummed queries. Under each bucket's name, the numbers in parentheses correspond to the k reference sequences selected in filter step and sampling parameter s used, respectively.

	Accuracy of ISMBGT								
	Synthetic					**Hummed**			
top-K	b_1 (3, 5)	b_2 (20, 3)	b_3 (18, 3)	b_4 (20, 5)	b_5 (10, 3)	b_1 (7, 3)	b_2 (2, 7)	b_3 (1, 7)	b_4 (10,5)
$K = 5$	18	76	158	119	60	3	7	15	7
$K = 10$	18	77	158	119	60	4	8	15	8
$K = 20$	18	78	158	119	61	5	9	16	8
$K = 50$	18	81	158	120	62	7	9	17	9
$K = 100$	19	84	158	120	63	9	9	17	11

Table 4 Accuracy (number of correct queries) of cDTW for synthetic and hummed queries. The Sakoe-Chiba band value for the synthetic and the hummed queries is 4 and 10, respectively.

	Accuracy of cDTW								
	Synthetic					**Hummed**			
top-K	b_1	b_2	b_3	b_4	b_5	b_1	b_2	b_3	b_4
$K = 5$	11	72	173	128	67	4	1	12	0
$K = 10$	15	74	174	129	67	5	4	13	0
$K = 20$	16	76	176	129	69	7	6	15	0
$K = 50$	16	79	177	129	69	8	12	20	0
$K = 100$	16	79	178	129	70	8	13	20	0

Regarding optimized cDTW, the average runtimes for the five buckets of the synthetic queries are 14.69, 30.79, 51.58, 95.42, and 104.34 seconds, respectively, and for the four buckets of the hummed queries they are 12.78, 21.52, 32.25, and 42.13 seconds, respectively. These runtimes (Table 5) indicate that for the case of synthetic queries, ISMBGT is up to 13 times faster than optimized cDTW for the synthetic queries and up to 6 times faster for the hummed queries. As a result, although cDTW provides better accuracies than ISMBGT for the synthetic queries (except for b_1) and comparable (but in general worse) results for the hummed queries, the runtime is much worse than that of ISMBGT.

6.2.5 Efficiency

In Table 7 we present the statistics for the efficiency evaluation measure per bucket for all queries. These statistics depend on the number of sequences evaluated at the refine step. As a result, we cannot easily compare the efficiency among different intervals.

Starting with the synthetic queries, we observe that the min values for all intervals range approximately from 2.61% (bucket b_3) to 3.36% (bucket b_1). The max values range from 9.59% (bucket b_2) to 18.62% (bucket b_5), showing that in the worst case we visit less than one fifth of the database and still manage to achieve high recall. Moreover, the mean and the median are very close to each other for all intervals, except for b_1 (mean is affected by max value). In addition, the me-

Table 5 Runtimes of brute-force SMBGT, ISMBGT, and optimized cDTW for the synthetic ('-S') and hummed queries ('-H') per bucket. *BF Ratio* is the ratio of times of BF to ISMBGT. The numbers in parentheses correspond to: k, reference sequences selected in the filter step and s, sampling parameter.

	Time (in seconds)				
Method	b_1	b_2	b_3	b_4	b_5
BF-S	7.13	13.46	20.90	29.30	45.08
ISMBGT-S	**3.57** (3, 5)	**5.99** (20, 3)	**6.51** (18, 3)	**6.84** (20, 5)	**7.71** (10, 3)
BF Ratio	1.99	2.24	3.20	4.28	5.84
Opt. cDTW	14.69	30.79	51.58	95.42	104.34
BF-H	16.31	28.76	38.96	62.18	-
ISMBGT-H	**5.99** (7, 3)	**3.67** (2, 7)	**8.70** (1, 7)	**7.47** (10, 5)	-
BF Ratio	2.72	7.82	4.47	8.32	-
Opt. cDTW	12.78	21.52	32.25	42.13	-

Table 6 Average times of ISMBGT for synthetic and hummed queries. The values of k and s per query length interval are given in Table 5.

	Query Length Intervals									
	Synthetic					**Hummed**				
ISMBGT Steps	b_1	b_2	b_3	b_4	b_5	b_1	b_2	b_3	b_4	b_5
Embedding	0.01	0.01	0.02	0.06	0.15	0.01	0.01	0.04	0.20	-
Filter	0.58	2.76	2.54	2.12	1.62	2.04	0.36	0.29	1.19	-
Refine	2.97	3.21	3.94	4.65	5.93	3.94	3.29	8.36	6.06	-

dian (and mean) increase when moving from b_1 to b_5, except for b_3 where the efficiency is a little higher than that of b_4. The mean efficiencies per interval are the following: 4.95%, 6.18%, 8.28%, 7.54%, and 10.71%. One reason for this is the fact that, for the intervals that involve larger queries, the SMBGT has to be evaluated for larger queries and segments of the database, while the number of sequences evaluated does not deviate much among intervals. These efficiency values show that there is a significant speedup of ISMBGT compared to BF as (approximately) at most only 10% of the database is actually evaluated with SMBGT.

Regarding the hummed queries, the minimum values for all intervals range from 3.80% (bucket b_2) to 6.77% (bucket b_3), while the maximum values from 6.74% (bucket b_1) to 40.84% (bucket b_3). Also, mean and median are very similar per bucket. The median efficiencies for the four buckets are the following: 5.75%, 4.43%, 20.49%, and 7.76%. These values are very low for b_1, b_2 and b_4, whereas for b_3 efficiency is 20.49%. This is due to the nature of the hummed queries of this interval; many more different subsequences are evaluated with SMBGT. Nonetheless, visiting on average one fifth of the database for these queries still shows the superiority of the proposed indexing approach over the brute-force search.

6.3 Highlights

Based on the results shown, we can first conclude that, for ISMBGT, the sampling parameter s can provide significant speedups for the filter step. ISMBGT can achieve recall values very close, or even higher for some cases, to those of BF by using relatively large sampling rates of up to $s = 7$, hence considering only $1/7^{th}$ of the embedding of the whole database.

In addition, although all results are based on randomly selecting reference sequences for all intervals, ISMBGT can achieve high recall values. It is plausible to expect that applying a more sophisticated method for selecting reference sequences can lead to even better accuracies. Still, even with randomly selected reference sequences, the experiments confirm that ISMBGT is meaningful and can be successfully applied to other time series domains.

Moreover, for synthetic queries the minimum gain we obtain with ISMBGT against BF in terms of runtime is 2 times (bucket b_1), while the maximum is 5.85 (bucket b_5). For hummed queries the minimum gain is 2.72 times (bucket b_1) and the maximum is 8.32 (bucket b_4). For queries in b_3 significant speedup can also be achieved (7.82 times). In general, for longer queries ISMBGT tends to obtain larger speedups with respect to BF. These values show the importance of ISMBGT, especially in real-time scenarios.

Table 7 Efficiency of ISMBGT for synthetic and hummed queries. The values of k and s per query length interval are given in Table 5.

	Query Length Intervals								
	Synthetic					**Hummed**			
% of |DB| evaluated	b_1	b_2	b_3	b_4	b_5	b_1	b_2	b_3	b_4
Min	3.35	3.06	2.60	3.29	2.70	5.19	3.80	6.77	6.01
Max	13.88	9.59	17.59	11.16	18.62	6.74	9.38	40.84	9.30
Mean	6.25	6.21	8.41	7.54	10.89	5.93	5.25	20.81	7.58
Median	4.94	6.18	8.28	7.54	10.71	5.74	4.43	20.49	7.76

Furthermore, comparing the runtimes of the three steps of ISMBGT we can conclude that the most demanding one is the refine step. The maximum time for the embedding step is 0.15 seconds for the synthetic (bucket b_5) and 0.21 seconds for the hummed queries (bucket b_4). For the filter step, time ranges from 0.58 to 2.76 seconds for the synthetic queries, and from 0.36 to 2.04 seconds for the hummed queries. The refine step requires at most 5.93 seconds for the synthetic queries (bucket b_5) and 6.06 seconds for the hummed ones (bucket b_4).

Finally, in terms of efficiency (for ISMBGT), the median is 4.94% (bucket b_1) and increases up to 10.71% (bucket b_5) for the synthetic queries, while for the hummed the lowest median is 4.43% (bucket b_2) and the highest 20.49% (bucket b_3). These results indicate that, in the worst case, only one fifth of the database will be (on average) evaluated using the costly SMBGT, making the proposed indexing method highly efficient for the QBH application domain.

7 Conclusions

Motivated by QBH we proposed ISMBGT, an embedding-based subsequence matching indexing approach, which allows for gaps in both query and target sequences, variable error tolerance, and constrains the maximum matching range and the minimum number of matched elements. The proposed filter-and-refine method can achieve speedups of up to an order of magnitude against brute-force SMBGT and and up to a factor of 13 for optimized cDTW. In addition, the recall values achieved by ISMBGT are very close to those of brute-force, even when using random reference sequences tailored for the given query. This is a substantial advantage over existing subsequence matching embedding-based methods. We plan to test ISMBGT in other noisy domains and investigate further ways of compressing the embedding space [22].

Acknowledgements

The work of I. Karlsson and P. Papapetrou was supported in part by the project "High-Performance Data Mining for Drug Effect Detection" funded by Swedish Foundation for Strategic Research under grant IIS11-0053. The work of V. Athitsos was partially supported by National Science Foundation grants IIS-0812601, IIS-1055062, CNS-1059235, CNS-1035913, and CNS-1338118. Finally, the work of D. Gunopulos was partially supported by the FP7-ICT project INSIGHT and the General Secretariat for Research and Technology ARISTEIA program project "MMD: Mining Mobility Data".

References

1. Athitsos, V., Alon, J., Sclaroff, S., Kollios, G.: BoostMap: A method for efficient approximate similarity rankings. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 268–275 (2004)
2. Athitsos, V., Hadjieleftheriou, M., Kollios, G., Sclaroff, S.: Query-sensitive embeddings. In: ACM International Conference on Management of Data (SIGMOD), pp. 706–717 (2005)
3. Bellman, R.: The theory of dynamic programming. Bull. Amer. Math. Soc. **60**(6), 503–515 (1954)
4. Bergroth, L., Hakonen, H., Raita, T.: A survey of longest common subsequence algorithms. In: SPIRE, pp. 39–48 (2000)
5. Bollobás, B., Das, G., Gunopulos, D., Mannila, H.: Time-series similarity problems and well-separated geometric sets. In: Symposium on Computational Geometry, pp. 454–456 (1997)
6. Chen, L., Ng, R.: On the marriage of l_p-norms and edit distance. In: VLDB, pp. 792–803 (2004)
7. Chen, L., Özsu, M.T.: Robust and fast similarity search for moving object trajectories. In: SIGMOD, pp. 491–502 (2005)
8. Chen, Y., Nascimento, M.A., Ooi, B.C., Tung, A.K.H.: Spade: On shape-based pattern detection in streaming time series. In: ICDE, pp. 786–795 (2007)
9. Crochemore, M., Iliopoulos, C., Makris, C., Rytter, W., Tsakalidis, A., Tsichlas, K.: Approximate string matching with gaps. Nordic Journal of Computing **9**(1), 54–65 (2002)
10. Dannenberg, R., Birmingham, W., Pardo, B., Hu, N., Meek, C., Tzanetakis, G.: A comparative evaluation of search techniques for query-by-humming using the

MUSART testbed. Journal of the American Society for Information Science and Technology **58**(5), 687–701 (2007)
11. Faloutsos, C., Lin, K.I.: FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In: ACM International Conference on Management of Data (SIGMOD), pp. 163–174 (1995)
12. Fu, A.W.c., Chan, P.M.s., Cheung, Y.L., Moon, Y.S.: Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances. The VLDB Journal **9**(2), 154–173 (2000). DOI 10.1007/PL00010672. URL `http://dx.doi.org/10.1007/PL00010672`
13. Fu, A.W.C., Keogh, E., Lau, L.Y.H., Ratanamahatana, C., Wong, R.C.W.: Scaling and time warping in time series querying. The Very Large DataBases (VLDB) Journal **17**(4), 899–921 (2008)
14. Han, T., Ko, S.K., Kang, J.: Efficient subsequence matching using the longest common subsequence with a dual match index. In: Machine Learning and Data Mining in Pattern Recognition, pp. 585–600 (2007)
15. Han, W.S., Lee, J., Moon, Y.S., Jiang, H.: Ranked subsequence matching in time-series databases. In: International Conference on Very Large Data Bases (VLDB), pp. 423–434 (2007)
16. Hjaltason, G., Samet, H.: Properties of embedding methods for similarity searching in metric spaces. IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI) **25**(5), 530–549 (2003)
17. Hristescu, G., Farach-Colton, M.: Cluster-preserving embedding of proteins. Tech. Rep. 99-50, CS Department, Rutgers University (1999)
18. Hu, N., Dannenberg, R., Lewis, A.: A probabilistic model of melodic similarity. In: ICMC, pp. 509–515 (2002)
19. Iliopoulos, C., Kurokawa, M.: String matching with gaps for musical melodic recognition. In: PSC, pp. 55–64 (2002)
20. Jang, J., Gao, M.: A query-by-singing system based on dynamic programming. In: International Workshop on Intelligent Systems Resolutions, pp. 85–89 (2000)
21. Keogh, E.: Exact indexing of dynamic time warping. In: International Conference on Very Large Databases (VLDB), pp. 406–417 (2002)
22. Keogh, E., Chu, S., Hart, D., Pazzani, M.: Segmenting time series: A survey and novel approach. In: In an Edited Volume, Data mining in Time Series Databases. Published by World Scientific, pp. 1–22. Publishing Company (1993)
23. Keogh, E., Pazzani, M.: Scaling up dynamic time warping for data mining applications. In: Proc. of SIGKDD (2000)
24. Kotsifakos, A., Papapetrou, P., Hollmén, J., Gunopulos, D.: A subsequence matching with gaps-range-tolerances framework: a query-by-humming application. Proceedings of VLDB **4**(11), 761–771 (2011)
25. Kotsifakos, A., Papapetrou, P., Hollmén, J., Gunopulos, D., Athitsos, V.: A survey of query-by-humming similarity methods. In: Proceedings of PETRA (2012)
26. Kotsifakos, A., Papapetrou, P., Hollmén, J., Gunopulos, D., Athitsos, V., Kollios, G.: Hum-a-song: a subsequence matching with gaps-range-tolerances query-by-humming system. Proceedings of the VLDB Endowment **5**(12), 1930–1933 (2012)
27. Kruskall, J.B., Liberman, M.: The symmetric time warping algorithm: From continuous to discrete. In: Time Warps. Addison-Wesley (1983)
28. Lemström, K., Ukkonen, E.: Including interval encoding into edit distance based music comparison and retrieval. In: AISB, pp. 53–60 (2000)
29. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics **10**(8), 707–710 (1966)
30. Maier, D.: The complexity of some problems on subsequences and supersequences. J. ACM **25**(2), 322–336 (1978)
31. Mongeau, M., Sankoff, D.: Comparison of musical sequences. Computers and the Humanities **24**(3), 161–175 (1990)
32. Papapetrou, P., Athitsos, V., Kollios, G., Gunopulos, D.: Reference-based alignment of large sequence databases. In: International Conference on Very Large Data Bases (VLDB) (2009)
33. Papapetrou, P., Athitsos, V., Potamias, M., Kollios, G., Gunopulos, D.: Embedding-based subsequence matching in time-series databases. ACM Transactions on Database Systems (TODS) **36**(3), 17 (2011)
34. Pardo, B., Birmingham, W.: Encoding timing information for musical query matching. In: ISMIR, pp. 267–268 (2002)
35. Pardo, B., Shifrin, J., Birmingham, W.: Name that tune: A pilot study in finding a melody from a sung query. Journal of the American Society for Information Science and Technology **55**(4), 283–300 (2004)
36. Park, S., Chu, W.W., Yoon, J., Won, J.: Similarity search of time-warped subsequences via a suffix tree. Information Systems **28**(7) (2003)
37. Park, S., Kim, S., Chu, W.W.: Segment-based approach for subsequence searches in sequence databases. In: ACM Symposium on Applied Computing (SAC), pp. 248–252 (2001)
38. Rabiner, L.: A tutorial on hidden Markov models and selected applications in speech recognition. Proceedings of the IEEE **77**(2), 257–286 (1989)
39. Rakthanmanon, T., Campana, B., Mueen, A., Batista, G., Westover, B., Zhu, Q., Zakaria, J., Keogh, E.: Searching and mining trillions of time series subsequences under dynamic time warping. In: Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 262–270. ACM (2012)
40. Sakoe, H., Chiba, S.: Dynamic programming algorithm optimization for spoken word recognition. Transactions on ASSP **26**, 43–49 (1978)
41. Sakurai, Y., Faloutsos, C., Yamamuro, M.: Stream monitoring under the time warping distance. In: ICDE, pp. 1046–1055 (2007)
42. Shou, Y., Mamoulis, N., Cheung, D.: Fast and exact warping of time series using adaptive segmental approximations. Machine Learning **58**(2-3), 231–267 (2005)
43. Uitdenbogerd, A., Zobel, J.: Melodic matching techniques for large music databases. In: ACM Multimedia (Part 1), p. 66 (1999)
44. Ukkonen, E., Lemström, K., Mäkinen, V.: Geometric algorithms for transposition invariant content-based music retrieval. In: ISMIR, pp. 193–199 (2003)
45. Unal, E., Chew, E., Georgiou, P., Narayanan, S.: Challenging uncertainty in query by humming systems: a fingerprinting approach. Transactions on Audio Speech and Language Processing **16**(2), 359–371 (2008)
46. Wang, X., Wang, J.T.L., Lin, K.I., Shasha, D., Shapiro, B.A., Zhang, K.: An index structure for data mining and clustering. Knowledge and Information Systems **2**(2), 161–184 (2000)
47. Zhou, M., Wong, M.: Efficient online subsequence searching in data streams under dynamic time warping distance. In: Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on, pp. 686–695. IEEE (2008)

48. Zhu, Y., Shasha, D.: Warping indexes with envelope transforms for query by humming. In: ACM International Conference on Management of Data (SIGMOD), pp. 181–192 (2003)