Data Min Knowl Disc
DOI 10.1007/s10618-009-0125-6

*i*SAX: disk-aware mining and indexing of massive time series datasets

Jin Shieh · Eamonn Keogh

Received: 16 July 2008 / Accepted: 19 January 2009

Abstract Current research in indexing and mining time series data has produced many interesting algorithms and representations. However, the algorithms and the size of data considered have generally not been representative of the increasingly massive datasets encountered in science, engineering, and business domains. In this work, we introduce a novel multi-resolution symbolic representation which can be used to index datasets which are several orders of magnitude larger than anything else considered in the literature. To demonstrate the utility of this representation, we constructed a simple tree-based index structure which facilitates fast exact search and orders of magnitude faster, approximate search. For example, with a database of one-hundred million time series, the approximate search can retrieve high quality nearest neighbors in slightly over a second, whereas a sequential scan would take tens of minutes. Our experimental evaluation demonstrates that our representation allows index performance to scale well with increasing dataset sizes. Additionally, we provide analysis concerning parameter sensitivity, approximate search effectiveness, and lower bound comparisons between time series representations in a bit constrained environment. We further show how to exploit the combination of both exact and approximate search as sub-routines in data mining algorithms, allowing for the exact mining of truly massive real world datasets, containing tens of millions of time series.

Responsible editor: Bart Goethals.

J. Shieh (✉) · E. Keogh
Department of Computer Science & Engineering, University of California, Riverside, CA, USA
e-mail: shiehj@cs.ucr.edu

E. Keogh
e-mail: eamonn@cs.ucr.edu

Keywords Time series · Data mining · Representations · Indexing

1 Introduction

The increasing level of interest in indexing and mining time series data has produced many algorithms and representations. However, with few exceptions, the size of datasets considered, indexed, and mined seems to have stalled at the megabyte level. At the same time, improvements in our ability to capture and store data have lead to the proliferation of terabyte-plus time series datasets. In this work, we show how a novel multi-resolution symbolic representation called *indexable Symbolic Aggregate approXimation* (*i*SAX) can be used to index datasets which are several orders of magnitude larger than anything else considered in current literature.

The *i*SAX approach allows for both fast exact search and ultra-fast approximate search. Beyond mere similarity search, we show how to exploit the combination of both types of search as sub-routines in data mining algorithms, permitting the exact mining of truly massive datasets, with tens of millions of time series, occupying up to a terabyte of disk space.

Our approach is based on a modification of the SAX representation to allow extensible hashing (Lin et al. 2007). That is, the number of bits used for evaluation of our representation can be dynamically changed, corresponding to a desired resolution. An increased number of bits can then be used to differentiate between non-identical entries. In essence, we show how we can modify SAX to be a multi-resolution representation, similar in spirit to wavelets (Chan and Fu 1999). It is this multi-resolution property that allows us to index time series with zero overlap at leaf nodes as in TS-tree (Assent et al. 2008), unlike R-trees (Guttman 1984), and other spatial access methods.

As we shall show, our indexing technique is fast and scalable due to intrinsic properties of the *i*SAX representation. Because of this, we do not require the use of specialized databases or file managers. Our results, conducted on massive datasets, are all achieved using a simple tree structure which uses the standard Windows XP NTFS file system for disk access. While it might have been possible to achieve faster times with a sophisticated DBMS, we feel that the simplicity of this approach is a great strength, and will allow easy adoption, replication, and extension of our work.

A further advantage of our representation is that, being symbolic, it allows the use of data structures and algorithms that are not well defined for real-valued data; including suffix trees, hashing, Markov models, etc. (Lin et al. 2007). Furthermore, given that *i*SAX is a superset of classic SAX, the several dozen research groups that use SAX will be able to adopt *i*SAX to improve scalability (Keogh 2008).

The rest of the paper is organized as follows. In Sect. 2 we review related work and background material. Section 3 introduces the *i*SAX representation, and Sect. 4 shows how it can be used for approximate and exact indexing. In Sect. 5 we perform a comprehensive set of experiments on both indexing and data mining problems. Finally, in Sect. 6 we offer conclusions and suggest directions for future work.

2 Background and related work

2.1 Time series distance measures

It is increasingly understood that Dynamic Time Warping (DTW) is better than Euclidean Distance (ED) for most data mining tasks in most domains (Xi et al. 2006). It is therefore natural to ask why we are planning to consider Euclidean distance in this work. The well documented superiority of DTW over ED is due to the fact that in small datasets it might be necessary to warp a little to match the nearest neighbor. However, in larger datasets one is more likely to find a close match without the need to warp. As DTW warps less and less, it degenerates to simple ED. This was first noted in Ratanamahatana and Keogh (2005) and later confirmed in Xi et al. (2006) and elsewhere. For completeness, we will show a demonstration of this effect. We measured the leave-one-out nearest neighbor classification accuracy of both DTW and ED on increasingly large datasets containing the CBF and Two-Pat problems, two classic time series benchmarks. Both datasets allow features to warp up to 1/8 the length of the sequence, so they may be regarded as highly warped datasets. Figure 1 shows the result.

As we can see, for small datasets, DTW is significantly more accurate than ED. However, as the datasets get larger, the difference diminishes, and by the time there are mere thousands of objects, there is no measurable difference. In spite of this, and for completeness, we explain in an online Appendix (Keogh and Shieh 2008) that we can index under DTW with *i*SAX with only trivial modifications.

2.2 Time series representations

There is a plethora of time series representations proposed to support similarity search and data mining. Table 1 shows the major techniques arranged in a hierarchy.

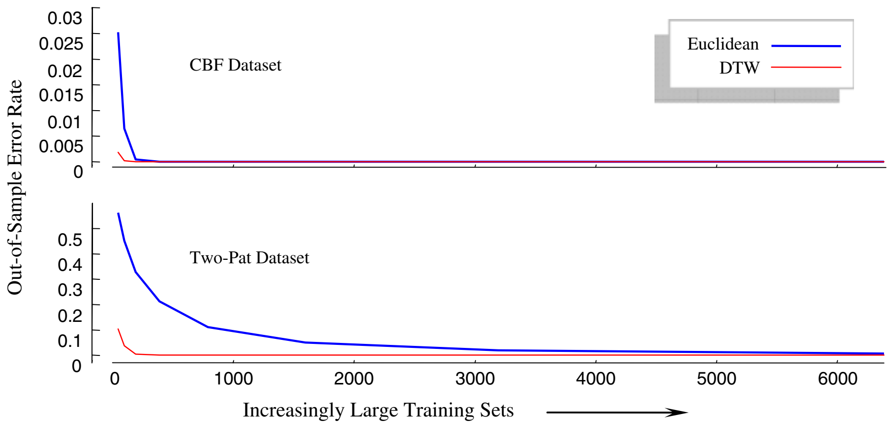

Fig. 1 The error rate of DTW and ED on increasingly large instantiations of the CBF and Two-Pat problems. For even moderately large datasets, there is no difference in accuracy

Table 1 A hierarchy of time series representations

Model based			
	Markov Models		
	Statistical Models		
	Time Series Bitmaps (Kumar et al. 2005)		
Data adaptive			
	Piecewise Polynomials		
		Interpolation* (Morinaka et al. 2001)	
		Regression (Shatkay and Zdonik 1996)	
	Adaptive Piecewise Constant Approximation* (Keogh et al. 2001b)		
	Singular Value Decomposition*		
	Symbolic		
		Natural Language (Portet et al. 2007)	
		Strings (Huang and Yu 1999)	
			Non-Lower Bounding (André-Jönsson and Badal 1997; Huang and Yu 1999; Megalooikonomou et al. 2005)
			SAX* (Lin et al. 2007), *i*SAX*
	Trees		
Non-data adaptive			
	Wavelets* (Chan and Fu 1999)		
	Random Mappings (Bingham and Mannila 2001)		
	Spectral		
		DFT* (Faloutsos et al. 1994)	
		DCT*	
		Chebyshev Polynomials* (Cai and Ng 2004)	
	Piecewise Aggregate Approximation* (Keogh et al. 2001a)		
	IPLA* (Chen et al. 2007)		
Data dictated			
	Clipped Data* (Bagnall et al. 2006)		

Those representations annotated with an asterisk have the very desirable property of allowing lower bounding. That is to say, we can define a distance measurement on the reduced abstraction that is guaranteed to be less than or equal to the true distance measured on the raw data. It is this lower bounding property that allows us to use a representation to index the data with a guarantee of no false dismissals (Faloutsos et al. 1994). The list of such representations includes (in approximate order of introduction) the discrete Fourier transform (DFT) (Faloutsos et al. 1994), the discrete Cosine transform (DCT), the discrete Wavelet transform (DWT), Piecewise Aggregate Approximation (PAA) (Keogh et al. 2001a), Adaptive Piecewise Constant Approximation (APCA), Chebyshev Polynomials (CHEB) (Cai and Ng 2004) and Indexable Piecewise Linear Approximation (IPLA) (Chen et al. 2007). We will provide the first empirical comparison of all these techniques in Sect. 5.

The only lower bounding omissions from our experiments are the eigenvalue analysis techniques such as SVD and PCA. While such techniques give optimal linear dimensionality reduction, we believe they are untenable for massive datasets. For example, while Steinbach et al. (2003) notes that they can transform 70,000 time series in under 10 min, this assumes the data can fit in main memory. However, to transform all the out-of-core (disk resident) datasets we consider in this work, SVD would require several months.

There have been several dozen research efforts that propose to facilitate time series search by first symbolizing the raw data (André-Jönsson and Badal 1997; Huang and Yu 1999; Megalooikonomou et al. 2005). However, in every case, the authors introduced a distance measure defined on the newly derived symbols. This allows false dismissals with respect to the original data. In contrast, the proposed work uses the symbolic words to internally organize and index the data, but retrieves objects with respect to the Euclidean distance on the original raw data. This point is important enough to restate. Although our proposed representation is an approximation to the original data, and whose creation requires us to make a handful of parameters choices, under *any* parameter set the exact search algorithm introduced in Table 6 is guaranteed to find the true exact nearest neighbor.

2.3 Review of classic SAX

The SAX representation was introduced in 2003, since then it has been used by more than 50 groups worldwide to solve a large variety of time series data mining problems (Lin et al. 2007; Keogh 2008). For concreteness, we begin with a review of it (Lin et al. 2007). In Fig. 2 (*left*) we illustrate a short time series T, which we will use as a running example throughout this paper.

Figure 2 (*right*) shows our sample time series converted into a representation called PAA (Keogh et al. 2001a). PAA represents a time series T of length n in a w-dimensional space by a vector of real numbers, $\bar{T} = \bar{t}_1, \ldots, \bar{t}_w$. The ith element of $\bar{T}$ is calculated by the equation:

$$\bar{t}_i = \frac{w}{n} \sum_{j=\frac{n}{w}(i-1)+1}^{\frac{n}{w}i} T_j$$

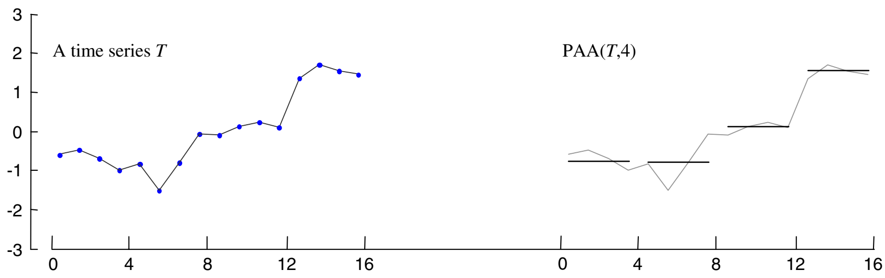

Fig. 2 (*left*) A time series T, of length 16. (*right*) A PAA approximation of T, with 4 segments

In the case that n is not divisible by w; the summation can be modified to adopt fractional values. This is illustrated in Lin et al. (2007).

PAA is a desirable intermediate representation as it allows for computationally fast dimensionality reduction, provides a distance measure which is lower bounding, and has been shown to be competitive with other dimensionality reduction techniques. In this case, the PAA representation reduces the dimensionality of the time series, from 16 to 4. The SAX representation takes the PAA representation as an input and discretizes it into a small alphabet of symbols with a cardinality of size **a**. The discretization is achieved by imagining a series of breakpoints running parallel to the x-axis and labeling each region between the breakpoints with a discrete label. Any PAA value that falls within that region can then be mapped to the appropriate discrete value.

While the SAX representation supports arbitrary breakpoints, we can ensure almost equiprobable symbols within a SAX word if we use a sorted list of numbers, $Breakpoints = \beta_1, \ldots, \beta_{a-1}$ such that the area under a $N(0,1)$ Gaussian curve from β_i to $\beta_{i+1} = 1/a$ (β_0 and β_a are defined as $-\infty$ and ∞, respectively). Table 2 shows a table for such breakpoints for cardinalities from 2 to 8.

A SAX word is simply a vector of discrete symbols. We use a boldface letter to differentiate between a raw time series and its SAX version, and we denote the cardinality of the SAX word with a superscript:

$$\mathrm{SAX}(T, w, a) = \mathbf{T^a} = \{t_1, t_2, \ldots, t_{w-1}, t_w\}$$

In previous work, we represented each SAX symbol as a letter or integer. Here however, we will use binary numbers for reasons that will become apparent later. For example, in Fig. 3 we have converted a time series T of length 16 to SAX words. Both examples have a word length of 4, but one has a cardinality of 4 and the other has a cardinality of 2. We therefore have $\mathrm{SAX}(T, 4, 4) = \mathbf{T^4} = \{\mathbf{11}, \mathbf{11}, \mathbf{01}, \mathbf{00}\}$ and $\mathrm{SAX}(T, 4, 2) = \mathbf{T^2} = \{\mathbf{1}, \mathbf{1}, \mathbf{0}, \mathbf{0}\}$.

The astute reader will have noted that once we have $\mathbf{T^4}$ we can derive $\mathbf{T^2}$ simply by ignoring the trailing bits in each of the four symbols in the SAX word. As one can readily imagine, this is a recursive property. For example, if we convert T to SAX with a cardinality of 8, we have $\mathrm{SAX}(T, 4, 8) = \mathbf{T^8} = \{\mathbf{110}, \mathbf{110}, \mathbf{011}, \mathbf{000}\}$. From this, we

Table 2 SAX breakpoints

β_i	a						
	2	3	4	5	6	7	8
β_1	0.00	−0.43	−0.67	−0.84	−0.97	−1.07	−1.15
β_2		0.43	0.00	−0.25	−0.43	−0.57	−0.67
β_3			0.67	0.25	0.00	−0.18	−0.32
β_4				0.84	0.43	0.18	0.00
β_5					0.97	0.57	0.32
β_6						1.07	0.67
β_7							1.15

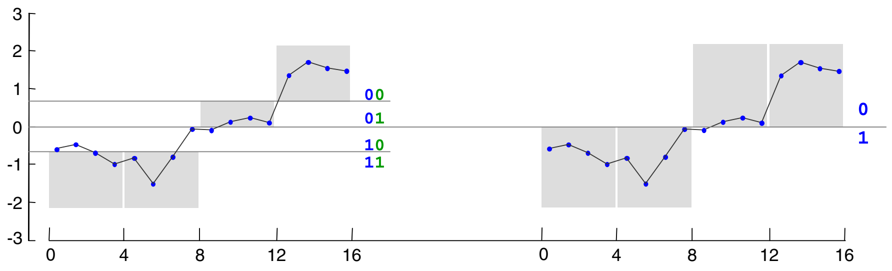

Fig. 3 A time series T converted into SAX words of cardinality 4 {**11, 11, 01, 00**} (*left*), and cardinality 2 {**1, 1, 0, 0**} (*right*)

Table 3 It is possible to obtain a reduced (by half) cardinality SAX word simply by ignoring trailing bits

SAX(T, 4, 16) = $\mathbf{T^{16}}$ = {**1100**, **1101**, **0110**, **0001**}
SAX(T, 4, 8) = $\mathbf{T^{8}}$ = {**110**, **110**, **011**, **000**}
SAX(T, 4, 4) = $\mathbf{T^{4}}$ = {**11**, **11**, **01**, **00**}
SAX(T, 4, 2) = $\mathbf{T^{2}}$ = {**1**, **1**, **0**, **0**}

can convert to any lower resolution that differs by a power of two, simply by ignoring the correct number of bits. Table 3 makes this clearer.

As we shall see later, this ability to change cardinalities on the fly is a useful and exploitable property.

Given two time series T and S, their Euclidean distance is:

$$D(T, S) \equiv \sqrt{\sum_{i=1}^{n} (T_i - S_i)^2}$$

If we have a SAX representation of these two time series, we can define a lower bounding approximation to the Euclidean distance as:

$$MINDIST\left(\mathbf{T^2}, \mathbf{S^2}\right) \equiv \sqrt{\frac{n}{w}} \sqrt{\sum_{i=1}^{w} (dist(t_i, s_i))^2}$$

This function requires calculating the distance between two SAX symbols and can be achieved with a lookup table, as in Table 4.

Table 4 A SAX *dist* lookup table for $a = 4$

	00	**01**	**10**	**11**
00	0	0	0.67	1.34
01	0	0	0	0.67
10	0.67	0	0	0
11	1.34	0.67	0	0

The distance between two symbols can be read off by examining the corresponding row and column. For example, $dist(\mathbf{00}, \mathbf{01}) = 0$ and $dist(\mathbf{00}, \mathbf{10}) = 0.67$.

For clarity, we will give a concrete example of how to compute this lower bound. Recall our running example time series T which appears in Fig. 2. If we create a time series S that is simply T's mirror image, then the Euclidean distance between them is $D(T, S) = 46.06$.

As we have already seen, $\text{SAX}(T, 4, 4) = \mathbf{T^4} = \{\mathbf{11}, \mathbf{11}, \mathbf{01}, \mathbf{00}\}$, and therefore $\text{SAX}(S, 4, 4) = \mathbf{S^4} = \{\mathbf{00}, \mathbf{01}, \mathbf{11}, \mathbf{11}\}$. The invocation of the *MINDIST* function will make calls to the lookup table shown in Table 4 to find:

$$\begin{aligned} dist(t_1, s_1) &= dist(\mathbf{11}, \mathbf{00}) = 1.34 \\ dist(t_2, s_2) &= dist(\mathbf{11}, \mathbf{01}) = 0.67 \\ dist(t_3, s_3) &= dist(\mathbf{01}, \mathbf{11}) = 0.67 \\ dist(t_4, s_4) &= dist(\mathbf{00}, \mathbf{11}) = 1.34 \end{aligned}$$

Which, when plugged into the *MINDIST* function, gives:

$$MINDIST\left(\mathbf{T^2}, \mathbf{S^2}\right) = \sqrt{\frac{16}{4}}\sqrt{1.34^2 + 0.67^2 + 0.67^2 + 1.34^2}$$

…to produce a lower bound value of 4.237. In this case, the lower bound is quite loose; however, having either more SAX symbols or a higher cardinality will produce a tighter lower bound. It is instinctive to ask how tight this lower bounding function can be, relative to natural competitors like PAA or DWT. This depends on the data itself and the cardinality of the SAX words, but coefficient for coefficient, it is surprisingly competitive with the other approaches. To see this, we can measure the *tightness of the lower bounds*, which is defined as the lower bounding distance over the true distance (Keogh et al. 2001a). Figure 4 shows this for random walk time series of length 256, with eight PAA or DWT coefficients and SAX words also of length eight. We varied the cardinality of SAX from 2 to 256, whereas PAA/DWT used a constant 4 bytes per coefficient. The results have been averaged over 10,000 random walk time series comparisons.

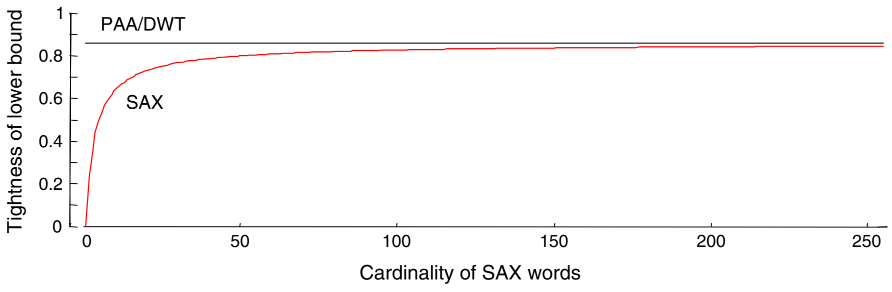

Fig. 4 The tightness of lower bounds for increasing SAX cardinalities, compared to a PAA/DWT benchmark

The results show that for small cardinalities the SAX lower bound is quite weak, but for larger cardinalities it rapidly approaches that of PAA/DWT. At the cardinality of 256, which take 8 bits, the lower bound of SAX is 98.5% that of PAA/DWT, but the latter requires 32 bits. This tells us that if we compare representations, coefficient for coefficient, there is little to choose between them; but if we do bit-for-bit comparisons (cf. Sect. 5), SAX allows for much tighter lower bounds. This is one of the properties of SAX that can be exploited to allow ultra-scalable indexing.

3 The *i*SAX representation

Because it is tedious to write out binary strings, previous uses of SAX had integers or alphanumeric characters representing SAX symbols (Lin et al. 2007). For example:

$$\text{SAX}(T, 4, 8) = \mathbf{T^8} = \{\mathbf{110}, \mathbf{110}, \mathbf{011}, \mathbf{000}\} = \{6, 6, 3, 0\}$$

However, this can make the SAX word ambiguous. If we see *just* the SAX word {6, 6, 3, 0} we cannot be sure what the cardinality is (although we know it is at least 7). Since all previous uses of SAX always used a single “hard-coded” cardinality, this has not been an issue. However, the fundamental contribution of this work is to show that SAX allows the comparison of words with different cardinalities, and even different cardinalities *within* a single word. We therefore must resolve this ambiguity. We do this by writing the cardinality as a superscript. For example, in the example above:

$$i\text{SAX}(T, 4, 8) = \mathbf{T^8} = \{6^8, 6^8, 3^8, 0^8\}$$

Because the individual symbols are ordinal, exponentiation is not defined for them, so there is no confusion in using superscripts in this context. Note that we are now using *i*SAX instead of SAX for reasons that will become apparent in a moment.

We are now ready to introduce a novel idea that will allow us to greatly expand the utility of *i*SAX.

3.1 Comparing different cardinality *i*SAX words

It is possible to compare two *i*SAX words of different cardinalities. Suppose we have two time series, *T* and *S*, which have been converted into *i*SAX words:

$$i\text{SAX}(T, 4, 8) = \mathbf{T^8} = \{\mathbf{110}, \mathbf{110}, \mathbf{011}, \mathbf{000}\} = \{6^8, 6^8, 3^8, 0^8\}$$
$$i\text{SAX}(S, 4, 2) = \mathbf{S^2} = \{\mathbf{0}, \mathbf{0}, \mathbf{1}, \mathbf{1}\} = \{0^2, 0^2, 1^2, 1^2\}$$

We can find the lower bound between *T* and *S*, even though the *i*SAX words that represent them are of different cardinalities. The trick is to *promote* the lower cardinality representation into the cardinality of the larger before giving it to the MINDIST function.

We can think of the tentatively promoted $\mathbf{S^2}$ word as $\mathbf{S^8} = \{\mathbf{0}_1^{**}, \mathbf{0}_2^{**}, \mathbf{1}_3^{**}, \mathbf{1}_4^{**}\}$, then the question is simply what are correct values of the missing $**_i$ bits? Note that both cardinalities can be expressed as the power of some integer. This guarantees an overlap in the breakpoints used during SAX computation. More concretely, if we have an *i*SAX cardinality of X, and an *i*SAX cardinality of 2X, then the breakpoints of the former are a proper subset of the latter. This is shown in Fig. 3.

Using this insight, we can obtain the missing bit values in $\mathbf{S^8}$ by examining each position and computing the bit values at the higher cardinality which are enclosed by the known bits at the current (lower) cardinality and returning the one which is closest in SAX space to the corresponding value in $\mathbf{T^8}$.

This method obtains the $\mathbf{S^8}$ representation usable for MINDIST calculations:

$$\mathbf{S^8} = \{\mathbf{011}, \mathbf{011}, \mathbf{100}, \mathbf{100}\}$$

It is important to note that this is *not* necessarily the same *i*SAX word we would have gotten if we had converted the original time series *S*. We cannot undo a lossy compression. However, using this *i*SAX word *does* give us an admissible lower bound.

Finally, note that in addition to comparing *i*SAX words of different cardinalities, the promotion trick described above can be used to compare *i*SAX words where *each* word has mixed cardinalities. For example, we can allow *i*SAX words such as $\{\mathbf{111}, \mathbf{11}, \mathbf{101}, \mathbf{0}\} = \{7^8, 3^4, 5^8, 0^2\}$. If such words exist, we can simply align the two words in question, scan across each pair of corresponding symbols, and promote the symbol with lower cardinality to the same cardinality as the larger cardinality symbol. In the next section, we explain why it is useful to allow *i*SAX words with different cardinalities.

4 *i*SAX indexing

4.1 The intuition behind *i*SAX indexing

As it stands, it may appear that the classic SAX representation offers the potential to be indexed. We could choose a fixed cardinality of, say, 8 and a word length of 4, and thus have 8^4 separate labels for files on disk. For instance, our running example *T* maps to $\{6^8, 6^8, 3^8, 0^8\}$ under this scheme, and would be inserted into a file that has this information encoded in its name, such as 6.8_6.8_3.8_0.8.txt. The query answering strategy would be very simple. We could convert the query into a SAX word with the same parameters, and then retrieve the file with that label from disk. The time series in that file are likely to be very good approximate matches to the query. In order to find the exact match, we could measure the distance to the best approximate match, then retrieve all files from disk whose label has a MINDIST value less than the value of the best-so-far match. Such a methodology clearly guarantees no false dismissals.

This scheme has a fatal flaw, however. Suppose we have a million time series to index. With 4,096 possible labels, the average file would have 244 time series in it, a reasonable number. However, this is the *average*. For all but the most contrived datasets we find a huge skew in the distribution, with more than half the files being empty, and the largest file containing perhaps 20% of the entire dataset. Either situation is undesirable for indexing, in the former case, if our query maps to an empty file, we would have to do some ad-hoc trick (perhaps trying "misspellings" of the query label) in order to get the first approximate answer back. In the latter case, if 20% of the data must be retrieved from disk, then we can be at most five times faster than sequential scan. Ideally, we would like to have a user defined threshold *th*, which is the maximum number of time series in a file, and a mapping technique that ensures each file has at least one and at most *th* time series in it. As we shall now see, *i*SAX allows us to guarantee exactly this.

*i*SAX offers a multi-resolution, bit aware, quantized, reduced representation with *variable* granularity. It is this variable granularity that allows us to solve the problem above. Imagine that we are in the process of building the index and have chosen $th = 100$. At some point there may be exactly 100 time series mapped to the *i*SAX word $\{2^4, 3^4, 3^4, 2^4\}$. If, as we continue to build the index, we find another time series maps here, we have an overflow, so we split the file. The idea is to choose one *i*SAX symbol, examine an additional bit, and use its value to create two new files. In this case:

Original File: $\{2^4, 3^4, 3^4, 2^4\}$ splits into…

$$\text{Childfile1} : \{4^8, 3^4, 3^4, 2^4\}$$
$$\text{Childfile2} : \{5^8, 3^4, 3^4, 2^4\}$$

Note that in this example we split on the first symbol, promoting the cardinality from 4 to 8. For some time series in the file, the extra bit in their first *i*SAX symbol was a **0**, and for others it was a **1**. In the former case, they are remapped to Child 1, and in the latter, remapped to Child 2. The child files can be named with some protocol that indicates their variable cardinality, for example **5.8**_3.4_3.4_2.4.txt and **4.8**_3.4_3.4_2.4.txt.

The astute reader will have noticed that the intuition here is very similar to the classic idea of extensible hashing. This in essence is the intuition behind building an *i*SAX index, although we have not explained *how* we decide which symbol is chosen for promotion and some additional details. In the next sections, we formalize this intuition and provide details on algorithms for approximately and exactly searching an *i*SAX index.

4.2 *i*SAX index construction

As noted above, a set of time series represented by an *i*SAX word can be split into two mutually exclusive subsets by increasing the cardinality along one or more dimensions. The number of dimensions d and word length, w, $1 \leq d \leq w$, provide an upper bound on the fan-out rate. If each increase in cardinality per dimension follows the assumption of iterative doubling, then the alignment of breakpoints contains overlaps in such a way that hierarchical containment is preserved between the common *i*SAX word and the set of *i*SAX words at the finer granularity. Specifically, in iterative doubling, the cardinality to be used after the ith increase in granularity is in accordance with the following sequence, given base cardinality $b : b{*}2^i$. The maximum fan-out rate under such an assumption is 2^d.

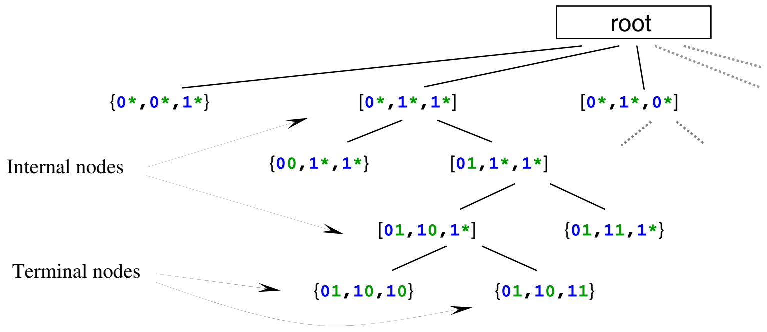

Fig. 5 An illustration of an *i*SAX index

The use of *i*SAX allows for the creation of index structures that are hierarchical, containing non-overlapping regions (Assent et al. 2008) (unlike R-trees etc., Faloutsos et al. 1994), and a controlled fan-out rate. For concreteness, we depict in Fig. 5 a simple tree-based index structure which illustrates the efficacy and scalability of indexing using *i*SAX.

The index is constructed given base cardinality b, word length w, and threshold th (b is optional; it can be defaulted to 2 or be set for evaluation to begin at higher cardinality). The index structure hierarchically subdivides the SAX space, resulting in differentiation between time series entries until the number of entries in each subspace falls below th. Such a construct is implemented using a tree, where each node represents a subset of the SAX space such that this space is a superset of the SAX space formed by the union of its descendents. A node's representative SAX space is congruent with an *i*SAX word and evaluation between nodes or time series is done through comparison of *i*SAX words. The three classes of nodes found in a tree and their respective functionality are described below:

4.2.1 Terminal node

A terminal node is a leaf node which contains a pointer to an index file on disk with raw time series entries. All time series in the corresponding index file are characterized by the terminal node's representative *i*SAX word. A terminal node represents the coarsest granularity necessary in SAX space to enclose the set of contained time series entries. In the event that an insertion causes the number of time series to exceed th, the SAX space (and node) is split to provide additional differentiation.

4.2.2 Internal node

An internal node designates a split in SAX space and is created when the number of time series contained by a terminal node exceeds th. The internal node splits the SAX space by promotion of cardinal values along one or more dimensions as per the iterative doubling policy. A hash from *i*SAX words (representing subdivisions of the SAX space) to nodes is maintained to distinguish differentiation between entries.

Time series from the terminal node which triggered the split are inserted into the newly created internal node and hashed to their respective locations. If the hash does not contain a matching *i*SAX entry, a new terminal node is created prior to insertion, and the hash is updated accordingly. For simplicity, we employ binary splits along a single dimension, using round robin to determine the split dimension.

4.2.3 Root node

The root node is representative of the complete SAX space and is similar in functionality to an internal node. The root node evaluates time series at base cardinality, that is, the granularity of each dimension in the reduced representation is *b*. Encountered *i*SAX words correspond to some terminal or internal node and are used to direct index functions accordingly. Un-encountered *i*SAX words during inserts result in the creation of a terminal node and a corresponding update to the hash table.

4.2.4 Index insertion

Pseudo-code of the insert function used for index construction is shown in Table 5. Given a time series to insert, we first obtain the *i*SAX word representation using the respective *i*SAX parameters at the current node (line 2). If the hash table does not yet contain an entry for the *i*SAX word, a terminal node is created to represent the relevant SAX space, and the time series is inserted accordingly (lines 22–24). Otherwise, there is an entry in the hash table, and the corresponding node is fetched. If this node is an

Table 5 *i*SAX index insertion function

```
1  Function Insert(ts)
2  iSAX_word = iSAX(ts, this.parameters)
3
4  if Hash.ContainsKey(iSAX_word)
5       node = Hash.ReturnNode(iSAX_word)
6       if node is terminal
7           if SplitNode() == false
8               node.Insert(ts)
9           else
10              newnode = new internal
11              newnode.Insert(ts)
12              foreach ts in node
13                  newnode.Insert(ts)
14              end
15              Hash.Remove(iSAX_word, node)
16              Hash.Add(iSAX_word, newnode)
17          endif
18      elseif node is internal
19          node.Insert(ts)
20      endif
21 else
22      newnode = new terminal
23      newnode.Insert(ts)
24      Hash.Add(iSAX_word, newnode)
25 endif
```

internal node, we call its insert function recursively (line 19). If the node is a terminal node, occupancy is evaluated to determine if an additional insert warrants a split (line 7). If so, a new internal node is created, and all entries enclosed by the overfilled terminal node are inserted (lines 10–16). Otherwise, there is sufficient space and the entry is simply added to the terminal node (line 8).

The deletion function is obvious and omitted for brevity.

4.3 Approximate search

For many data mining applications, an approximate search may be all that is required. An *i*SAX index is able to support very fast approximate searches; in particular, they only require a single disk access. The method of approximation is derived from the intuition that two similar time series are often represented by the same *i*SAX word. Given this assumption, the approximate result is obtained by attempting to find a terminal node in the index with the same *i*SAX representation as the query. This is done by traversing the index in accordance with split policies and matching *i*SAX representations at each internal node. Because the index is hierarchical and without overlap, if such a terminal node exists, it is promptly identified. Upon reaching this terminal node, the index file pointed to by the node is fetched and returned. This file will contain at least 1 and at most *th* time series in it. A main memory sequential scan over these time series gives the approximate search result.

In the (very) rare case that a matching terminal node does not exist, such a traversal will fail at an internal node. We mitigate the effects of non-matches by proceeding down the tree, selecting nodes whose last split dimension has a matching *i*SAX value with the query time series. If no such node exists at a given junction, we simply select the first, and continue the descent.

4.4 Exact search

Obtaining the exact nearest neighbor to a query is both computationally and I/O intensive. To improve search speed, we use a combination of approximate search and lower bounding distance functions to reduce the search space. The algorithm for obtaining the nearest neighbor is presented as pseudo-code in Table 6.

The algorithm begins by obtaining an approximate best-so-far (BSF) answer, using approximate search as described in Sect. 4.3 (lines 2–3). The intuition is that by quickly obtaining an entry which is a close approximation and with small distance to the nearest neighbor, large sections of the search space can be pruned. Once a baseline BSF is obtained, a priority queue is created to examine nodes whose distance is potentially less than the BSF. This priority queue is first initialized with the root node (line 6).

Because the query time series is available to us, we are free to use its PAA representation to obtain a tighter bound than the MINDIST between two *i*SAX words. More concretely, the distance used for priority queue ordering of nodes is computed using

Table 6 Expediting exact search using approximate search and lower bounding distance functions

```
1   Function [IndexFile] = ExactSearch(ts)
2   BSF.IndexFile = ApproximateSearch(ts)
3   BSF.dist = IndexFileDist(ts, BSF.IndexFile)
4
5   PriorityQueue pq
6   pq.Add(root)
7
8   while !pq.IsEmpty
9       min = pq.ExtractMin()
10      if min.dist >= BSF.dist
11          break
12      endif
13      if min is terminal
14          tmp = IndexFileDist(ts, min.IndexFile)
15          if BSF.dist > tmp
16              BSF.dist = tmp
17              BSF.IndexFile = min.IndexFile
18          endif
19      elseif min is internal or root
20          foreach node in min.children
21              node.dist = MINDIST_PAA_iSAX(ts,node.iSAX)
22              pq.Add(node)
23          end
24      endif
25  end
26  return BSF.IndexFile
```

MINDIST_PAA_*i*SAX, between the PAA representation of the query time series and the *i*SAX representation of the SAX space occupied by a node.

Given the PAA representation, $\mathrm{T}_{\mathrm{PAA}}$ of a time series T and the *i*SAX representation, $\mathrm{S}_{i\mathrm{SAX}}$ of a time series S, such that $|\mathrm{T}_{\mathrm{PAA}}| = |\mathrm{S}_{i\mathrm{SAX}}| = w$, $|T| = |S| = n$, and recalling that the jth cardinal value of $\mathrm{S}_{i\mathrm{SAX}}$ derives from a PAA value, v between two breakpoints $\beta_L, \beta_U, \beta_L < v \le \beta_U, 1 \le j \le w$ we define the lower bounding distance as:

$$\mathrm{MINDIST\_PAA\_}i\mathrm{SAX}\,(\mathrm{T}_{\mathrm{PAA}}, \mathrm{S}_{i\mathrm{SAX}}) = \sqrt{\frac{n}{w}}\sqrt{\sum_{i=1}^{w} \begin{cases} (\beta_{Li} - T_{PAAi})^2 \; if \;\; \beta_{Li} > T_{PAAi} \\ (\beta_{Ui} - T_{PAAi})^2 \; if \;\; \beta_{Ui} < T_{PAAi} \\ 0 \;\; otherwise \end{cases}}$$

Recall that we use distance functions that lower bound the true Euclidean distance. That is, if the BSF distance is less than or equal to the minimum distance from the query to a node, we can discard the node and all descendants from the search space without examining their contents or introducing any false dismissals.

The algorithm then repeatedly extracts the node with the smallest distance value from the priority queue, terminating when either the priority queue becomes empty or an early termination condition is met. Early termination occurs when the lower bound distance we compute equals or exceeds the distance of the BSF. This implies that the remaining entries in the queue cannot qualify as the nearest neighbor and can be discarded.

If the early termination condition is not met (line 10), the node is further evaluated. In the case that the node is a terminal node, we fetch the index file from disk and compute the distance from the query to each entry in the index file, recording the minimum distance (line 14). If this distance is less than our BSF, we update the BSF (lines 16–17).

In the case that the node is an internal node or the root node, its immediate descendents are inserted into the priority queue (lines 20–23). The algorithm then repeats by extracting the next minimum node from the priority queue.

Before leaving this section, we note that we have only discussed 1NN queries. Extensions to KNN and range queries are trivial and obvious, and are omitted for brevity.

4.4.1 Extension with time series wedges

Extensions to the index are readily facilitated as meta-information can be held in nodes. This allows the index to supplant or be used in concert with external techniques. For experimental purposes, and to expedite exact search, we modified index terminal nodes by storing meta-data which are used to obtain a lower bounding distance to the set of contained time series at each terminal node. This distance is a potentially tighter bound than that of MINDIST_PAA_*i*SAX.

Specifically, terminal nodes in the index now maintain a record of the minimum and maximum value per dimension from the set of contained time series as an upper and lower wedge, a technique described in Wei et al. (2005) and illustrated in Fig. 6. Given that a terminal node in a non-trivial tree is essentially a grouping of similar time series, we expect the wedges to be tight; making them an advantageous addition for search space pruning. When exact search encounters a terminal node and early termination conditions are not met, we compute a second lower bounding distance using LB_Keogh (Wei et al. 2005) from the recorded wedges. As the upper and lower wedge is saved as meta-data in each terminal node, the LB_Keogh computation does not require additional disk accesses. If this distance is greater or equal to the BSF, we can safely discard the terminal node from consideration without fetching its index file from disk. Given that repeated disk accesses can become prohibitively expensive; the addition of wedges has significant utility.

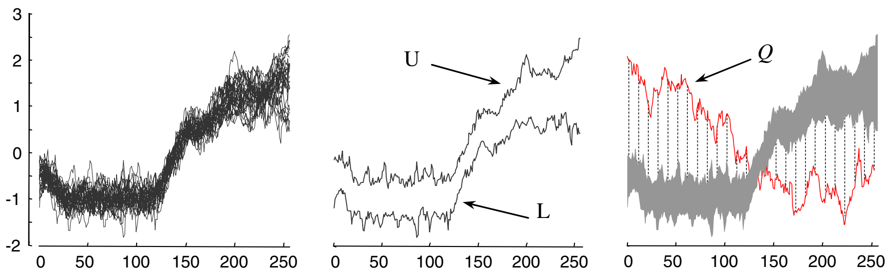

Fig. 6 (*left*) A set of time series which all map to the *i*SAX word $\{2^8, 1^8, 1^8, 1^8, 1^8, 1^8, 1^8, 1^8, 3^8, 5^8, 2^4, 3^4, 3^4, 3^4, 3^4, 3^4\}$. (*center*) The maximum/minimum values for the set can be used to define upper/lower wedges. (*right*) The square root of the sum of squared lengths of the hatch lines is a lower bound to the Euclidean between Q and any within the set

5 Experiments

We begin by discussing our experimental philosophy. We have designed all experiments such that they are not only reproducible, but *easily* reproducible. To this end, we have built a webpage which contains all datasets used in this work, together with spreadsheets which contain the raw numbers displayed in all the figures (Keogh and Shieh 2008). In addition, the webpage contains many additional experiments which we could not fit into this work; however, we note that this paper is completely self-contained.

We have used random walk datasets for much of our experimental work because it is known to model stock market data very well, and for the simple pragmatic reason that it is easy to create arbitrarily large datasets, which can be exactly recreated by others who only need to know the seed value. We note, however, that in this work we also test on heartbeat and insect data, which are very different from random walks, and in the website built to support this work we show results on 30 diverse datasets.

Experiments are conducted on an AMD Athlon 64 X2 5600+ with 3GB of memory, Windows XP SP2 with /3GB switch enabled, and using version 2.0 of the .NET Framework. All experiments used a 400GB Seagate Barracuda 7200.10 hard disk drive with the exception of the 100M random walk experiment, which required additional space, there we used a 750GB Hitachi Deskstar 7K10000.

5.1 Tightness of lower bounds

It is important to note that the rivals to *i*SAX are other time series representations, not indexing structures such as R-Trees, VP-Trees, etc. (Ding et al. 2008). We therefore begin with a simple experiment to compare the tightness of lower bounds of *i*SAX with the other lower bounding time series representations, including DFT, DWT, DCT, PAA, CHEB, APCA and IPLA. We measure TLB, the tightness of lower bounds (Keogh et al. 2001a). This is calculated as:

$$\mathrm{TLB} = \mathrm{LowerBoundDist}\,(T, S)/\mathrm{TrueEuclideanDist}\,(T, S)$$

Because DWT and PAA have exactly the same TLB (Keogh et al. 2001a) we show one graphic for both. We randomly sample T and S (with replacement) 1,000 times for each combination of parameters. We vary the time series length [480, 960, 1440, 1920] and the number of bytes per time series available to the dimensionality reduction approach [16, 24, 32, 40]. We assume that each real valued representation requires 4 bytes per coefficient, thus they use [4, 6, 8, 10] coefficients. For *i*SAX, we hard code the cardinality to 256, resulting in [16, 24, 32, 40] symbols per word.

Recall that, for TLB, larger values are better. If the value of TLB is zero, then any indexing technique is condemned to retrieving every object from the disk. If the value of TLB is one, then there is no search, we could simply retrieve one object from disk and guarantee that we had the true nearest neighbor. Figure 7 shows the result of one such experiment with an ECG dataset.

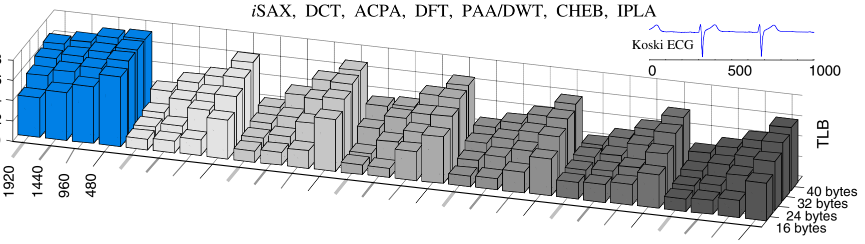

Fig. 7 The tightness of lower bounds for various time series representations on the Koski ECG dataset. Similar graphs for thirty additional datasets can be found at Keogh and Shieh (2008)

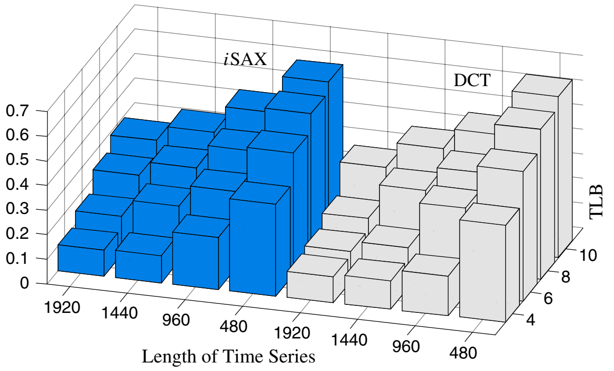

Fig. 8 The experiment in the previous figure redone with the *i*SAX word length equal to the dimensionality of the real valued applications (just DCT is shown to allow a "zoom in")

Note that the speedup obtained is generally non-linear in TLB, that is to say if one representation has a lower bound that is twice as large as another, we can usually expect a *much* greater than two-fold decrease in disk accesses.

In a sense, it may be obvious before doing this experiment that *i*SAX will have a smaller reconstruction error, thus a tighter lower bound, and greater indexing efficiency than the real valued competitors. This is because *i*SAX is taking advantage of every bit given to it. In contrast, for the real valued approaches it is clear that the less significant bits contribute *much* less information than the significant bits. If the raw time series is represented with 4 bytes per data point, then each real valued coefficient must also have 4 bytes (recall that orthonormal transforms are merely rotations in space). This begs the question, why not quantize or truncate the real valued coefficients to save space? In fact, this is a very common idea in compression of time series data. For example, in the medical domain it is frequently done for both the wavelet (Chen and Itoh 1998) and cosine (Batista et al. 2001) representations. However, recall that we are not interested in compression per se. Our interest is in dimensionality reduction that allows indexing with no false dismissals. If, for the other approaches, we save space by truncating the less significant bits, then at least under the IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754) default policy for rounding (RoundtoNearest) it is possible the distance between two objects can *increase*, thus violating the no false dismissals guarantee. We have no doubt that an indexable bit-adjustable version of the real valued representations could be made to work, however, none exists to date.

Even if we naively coded each *i*SAX word with the same precision as the real valued approaches (thus wasting 75% of the main memory space), *i*SAX is still competitive with the other approaches; this is shown in Fig. 8. Before leaving this section, we note that we have repeated these experiments with thirty additional datasets from very diverse domains with essentially the same results (Keogh and Shieh 2008).

5.2 Sensitivity to parameters

For completeness, we conduct experiments which evaluate the sensitivity of *i*SAX indexing to parameter values. Recall that an *i*SAX index is constructed given the following: base cardinality b, word length w, and a threshold *th* (the maximum number of entries in a leaf node). Our analysis focuses on the parameters w and *th*. We exclude the evaluation of b, which is used during computation of new cardinal values, as this is a procedure which inherently conforms to the size and skew of the indexed data. For b, any low value will suffice. The following analysis identifies the characteristics of an *i*SAX index containing 1 million random walk time series of length 256 from the averaged results of 1000 approximate queries, with respect to a range of *th* and w values. The quality of index performance can be gauged by consideration of both the number of index files created as well as the rank of approximate search results. For approximate search rankings, we measure the percentage of queries which returns an entry which is the true nearest neighbor, an entry which ranks within the top 10 nearest neighbors, an entry which ranks within the top 100 nearest neighbors, and an entry which ranks outside the top 1000 nearest neighbors. Increases in the first three measures or a decrease in the final, indicate a favorable trend with respect to the quality of index results. The experimental analysis below validates our choice of parameter values used in later sections.

In Figs. 9 and 10 we vary the *th* value between [25, 50, 100, 200, 400, 800, 1600] while keeping w and b stationary at 8 and 4, respectively. As illustrated by the gradually sloped curves in Fig. 9, index performance is not sharply affected by *th* values. Therefore, the determination of an adequate *th* value rests on the tradeoff between possible entries retrieved (*th*), and the number of index files created. Our choice of

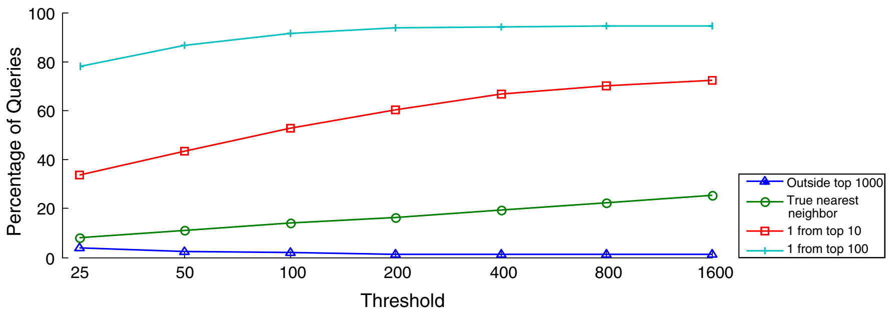

Fig. 9 Approximate search rankings for increasing threshold values

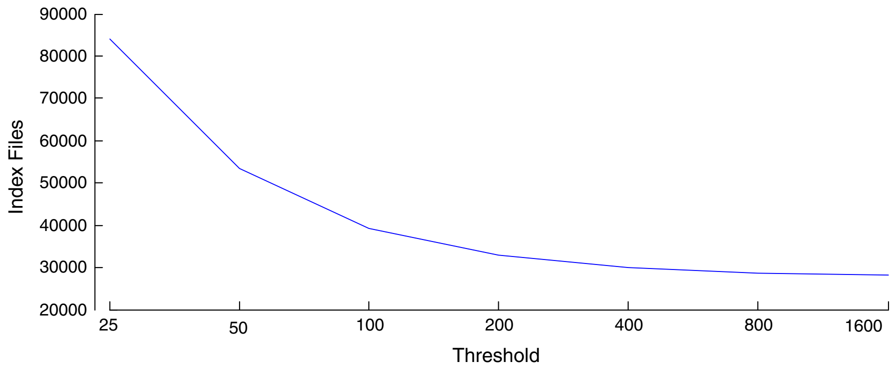

Fig. 10 Index files created across varying threshold values

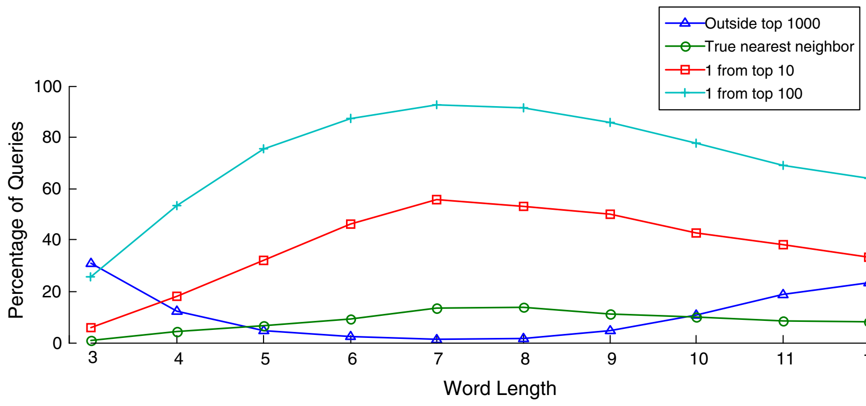

Fig. 11 Approximate search rankings for increasing word length

th = 100 in Sect. 5.3 is affirmed as a suitable choice as this is characterized by both a low number of entries examined and by having the number of index files created approach the bottom of the curve in Fig. 10.

In Figs. 11 and 12, we vary the value of w between [3–12] while keeping *th* and b stationary at 100 and 4, respectively. The results indicate that index performance is not highly dependent on the selection of very precise w values. In Fig. 11, there exists a range of values, [6–9], where approximate search rankings maintain a high level of performance. We observe some degradation in performance with increasingly longer word lengths, though this is expected as smaller segments result in increased sensitivity to noise. We also examined the number of index files created and showed that this number increases with w (though for low values of w, there may be a minimum number of index files necessary to support the dataset, given *th*). This increase in index files is an expected trend, as an increase in w corresponds to an increase in the set of possible *i*SAX words (which are used for index filenames). Our analysis affirms our choice of $w = 8$ in Sect. 5.3 as a suitable value, falling in the range of w which returns

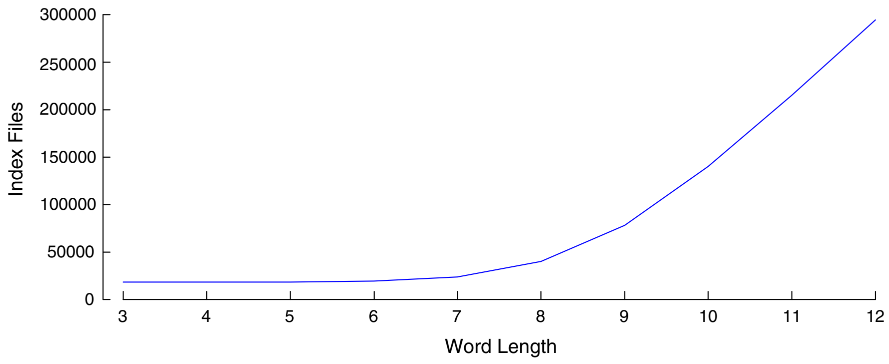

Fig. 12 Index files created across varying word lengths

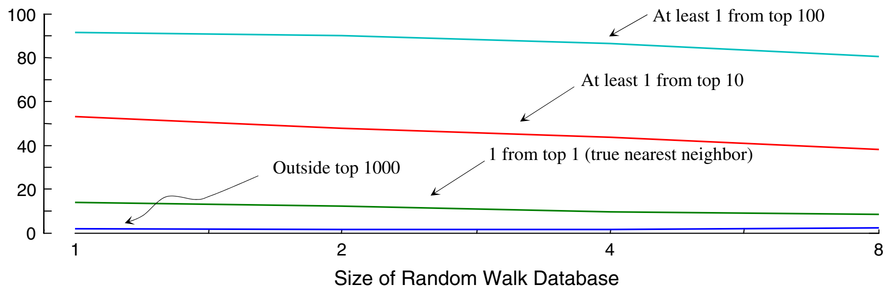

Fig. 13 The percentage of cutoffs for various rankings, for increasingly large databases with approximate search

quality results while also at the lower end of the spectrum with regards to index files created.

Our analysis of index characteristics across a range of parameter values have shown that parameters should be selected in consideration of both search performance as well as the number of index files constructed. The selections of these key parameters, while critical, have been shown to be generally flexible and competitive across a range of values and without the need for exact tuning.

5.3 Indexing massive datasets

We tested the accuracy of approximate search for increasingly large random walk databases of sequence length 256, containing [one, two, four, eight] million time series. We used $b = 4$, $w = 8$, and $th = 100$. This created [39,255; 57,365; 92,209; 162,340] files on disk. We generated 1,000 queries, did an approximate search, and then compared the results with the true ranking which we later obtained with a sequential scan. Figure 13 shows the results.

The figure tells us that when searching one million time series, 91.5% of the time approximate search returns an answer that would rank in the top 100 of the true nearest neighbor list. Furthermore, that percentage only slightly decreases as we scale to eight million time series. Likewise, again, for one million objects, more than half the time the approximate searches return an object that would rank in the top 10, and 14% of the time it returns the *true* nearest neighbor. Recall that these searches require exactly one disk access and at most 100 Euclidean distance calculations, so the average time for a query was less than a second.

We also conducted exact search experiments on 10% of the queries. Figure 14 shows the estimated wall clock time and Fig. 15 shows the average disk I/O for exactly finding the nearest neighbor using the *i*SAX index. Sequential scan is used as a baseline for comparison.

To push the limits of indexing, we considered indexing 100,000,000 random walk time series of length 256. To the best of our knowledge, this is as least two orders

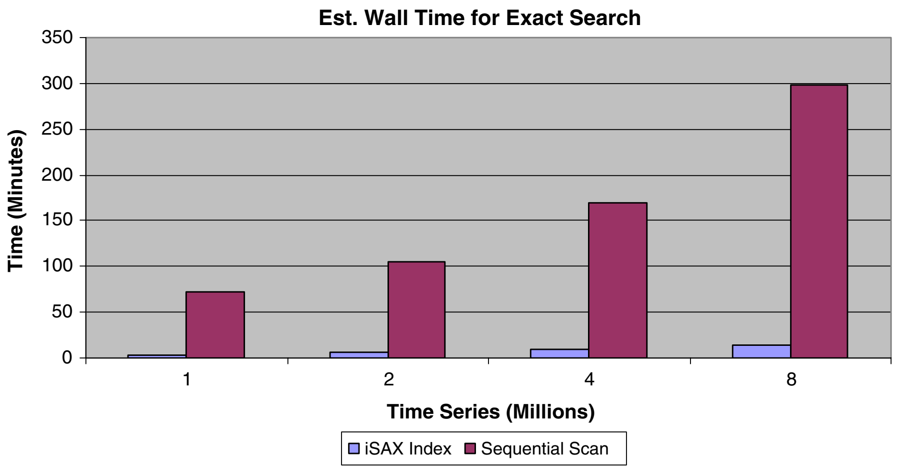

Fig. 14 Estimated wall clock time for exact search averaged over 100 queries

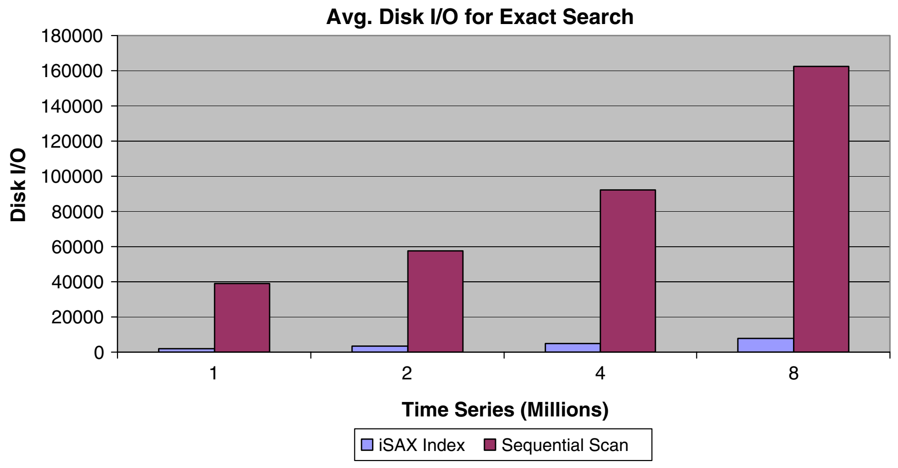

Fig. 15 Average disk I/O for exact search averaged over 100 queries

of magnitude larger that any other dataset considered in the literature (Assent et al. 2008; Cai and Ng 2004; Faloutsos et al. 1994; Megalooikonomou et al. 2005). Since the publication of *Don Quixote de la Mancha* in the 17th century, the idiom, "*a needle in a haystack*" has been used to signify a near impossible search. If each time series in this experiment was represented by a piece of hay the size of a drinking straw, they would form a cube shaped haystack with 262 meter sides.

Because of the larger size of data, we increased *th* to 2,000, and used w of 16. This created 151,902 files occupying a half terabyte of disk space. The average occupancy of index files is approximately 658.

We issued ten new random walk approximate search queries. Each query was answered in an average of 1.15 s. To find out how good each answer was, we did a linear scan of the data to find the true rankings of the answers. Three of the queries did actually discover their true nearest neighbor, the average rank was 8, and the worst query "only" managed to retrieve its 25th nearest neighbor. In retrospect, these results are extraordinarily impressive. Faced with one hundred million objects on disk, we can retrieve only 0.0013895% of the data and find an object that is ranked the top 0.0001%. As we shall see in Sects. 5.5 and 5.6, the extraordinary precision and speed of approximate search combined with fast exact search allows us to consider mining datasets with millions of objects.

We also conducted exact searches on this dataset; each search took an average of 90 min to complete, in contrast to a linear scan taking 1,800 min.

5.4 Approximate search evaluation

Approximate search, being orders of magnitude faster than exact search, is inherently attractive for many problems. Because the returned results are approximate in nature, it is necessary for us to ascertain the general quality and effectiveness of said results. We have seen in Sect. 5.3 some measure of this and we reaffirm its utility here with additional visual and quantitative evaluations.

In the arid to semi-arid regions of North America, the Beet leafhopper (*Circulifer tenellus*) is the only known vector (carrier) of curly top virus, which causes major economic losses in a number of crops including sugarbeet, tomato, and beans Kaffka et al. (2000). In order to mitigate these financial losses, entomologists at the University of California, Riverside are attempting to model and understand the behavior of this insect. It is known that the insects feed by sucking sap from living plants; much like how mosquitoes suck blood from mammals and birds. In order to understand the insect's behaviors, entomologists' glue a thin wire to the insect's back, complete the circuit through a host plant, and then measure fluctuations in voltage level to create an Electrical Penetration Graph (EPG), a time series, which can then be mined for clues to insect behavior. The problem facing the entomologists is that these experiments have left them with massive data collections which are difficult to search.

We indexed the entire insect data archive of 4,232,591 subsequences of length 150 using $b = 4$, $w = 8$, $th = 100$.

We asked the entomologist Dr. Greg Walker to draw a query time series. He was interested in knowing if the database contained any examples of a pattern called

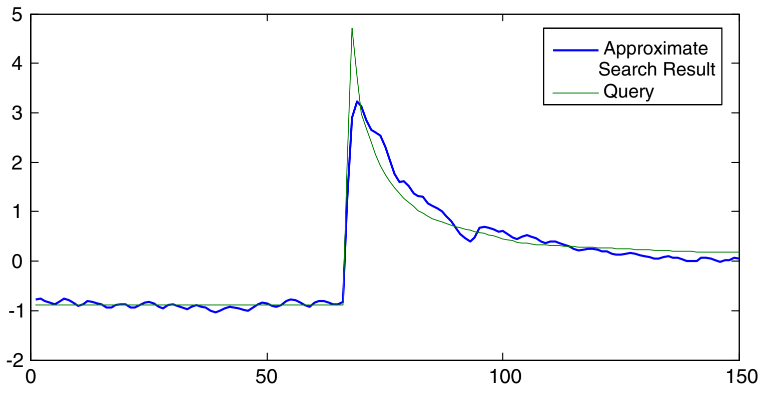

Fig. 16 Approximate search result on insect dataset

"Waveform **A**", which he noted is characterized by "*an almost vertical increase in the voltage level from baseline. Immediately after this spike, there is a gradual decline in voltage level which occurs as a smooth downward curve*". This pattern is produced during the initial penetration of the plant tissue through the epidermis.

The idealized query time series and corresponding approximate search result is shown in Fig. 16. As shown by the figure, a simple approximate search is capable of retrieving a matching shape and corresponding location to researchers for further analysis. Although this experiment searched a database of over four million time series, the result was returned in less than a second, allow rapid interaction and hypothesis testing for the scientist.

Section 5.3 identified characteristics of approximate search in the form of nearest neighbor rankings. In this section, we quantify the effectiveness of approximate search results via comparison with actual nearest neighbors. We indexed 9,999,745 random walk time series subsequences of length 256 with parameters $b = 4$, $w = 8$, $th = 150$. 100 random walk queries were generated and the approximate search result of each was obtained. To quantify the quality of approximate search results we formulate a distance ratio which compares the true nearest neighbor distance and the approximate search distance. Let time series Q, A, T be the query, the approximate result, and the true nearest neighbor, respectively. Calculate:

$$\text{DistanceRatio} = \text{EuclideanDist}\,(Q, T)/\,\text{EuclideanDist}\,(Q, A)$$

This distance ratio is an indicator of how similar the approximate result is, relative to that of the true nearest neighbor. Figure 17 shows the distance ratio for each of the 100 queries, sorted in ascending order. All ratios are above 0.69, which indicates no approximate result deviates significantly from the actual nearest neighbor. For additional illustration on the quality of approximate results, the Q, A, T set of time series corresponding to the lower median of distance ratios (0.907) is shown in Fig. 18. In fact, it is extremely hard to make any visual determination as to which plot is the approximate result and which corresponds to the actual nearest neighbor (without the aid of a legend).

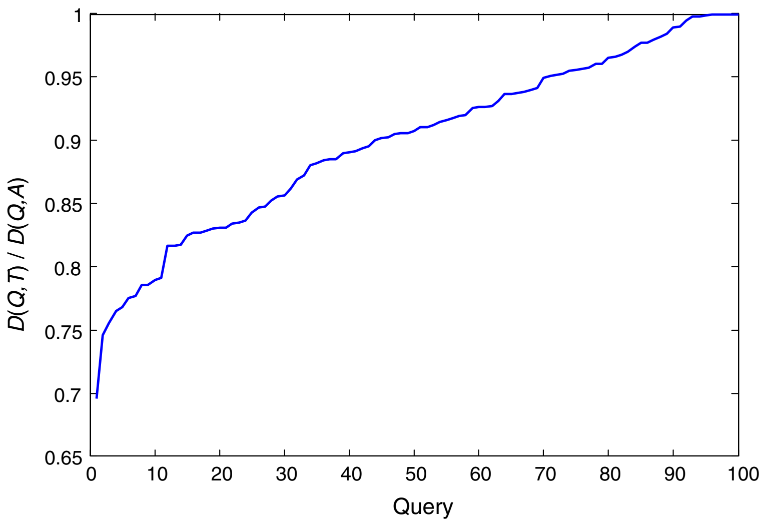

Fig. 17 Sorted distance ratios of 100 random walk queries

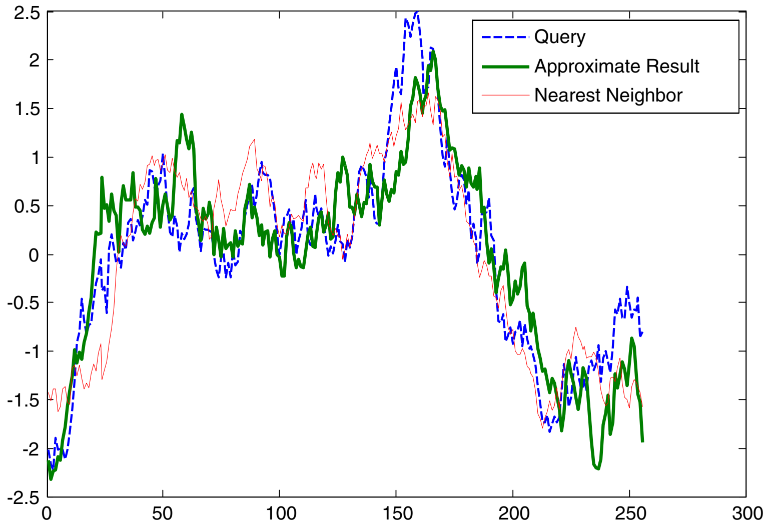

Fig. 18 Plot showing the random walk query, approximate result, and true nearest neighbor. Together they correspond to the lower median of distance ratios computed

5.5 Time series set difference

In this section, we give an example of a data mining algorithm that can be built on top of our existing indexing algorithms. The algorithm is interesting in that it uses both approximate search and exact search to compute the ultimate (exact) answer to a problem.

Suppose we are interested in contrasting two collections of time series data. For example, we may be interested in contrasting telemetry from the last Shuttle launch with telemetry from all previous launches, or we may wish to contrast the 10 min of electrocardiograms just before a patient wakes up with the preceding seven hours of sleep.

To do this, we define the Time Series Set Difference (TSSD):

Definition Time Series Set Difference (*A*,*B*). Given two collections of time series *A* and *B*, the time series set difference is the time series in *A* whose distance from its nearest neighbor in *B* is maximal.

Note that we are not claiming that this is the best way to contrast two sets of time series; it is merely a sensible definition we can use as a starting point.

We tested this definition on an electrocardiogram dataset. The data is an overnight polysomnogram with simultaneous three-channel Holter ECG from a 45 year old male subject with suspected sleep-disordered breathing. We used the first 7.2 h of the data as the reference set *B*, and the next 8 min 39 s as the "novel" set *A*. The set *A* corresponds to the period in which the subject woke up. After indexing the data with an *i*SAX word length of 9 and a maximum threshold value of 100, we had 1,000,000 time series subsequences in 31,196 files on disk, occupying approximately 4.91GB of secondary storage. Figure 19 show the TSSD discovered.

We showed the result to UCLA cardiologist Helga Van Herle. She noted that the p-waves in each of the full heartbeats look the same, but there is a 21.1% increase in the length of the second one. This indicated to her that this is almost certainly an example of sinus arrhythmia, where the R-R intervals are changing with the patients breathing pattern. This is likely due to slowing of the heart rate with expiration and increase of the heart rate with inspiration, given that it is well known that respiration patterns change in conjunction with changes in sleep stages (Scholle and Schäfer 1999).

An obvious naive algorithm to find the TSSD is to do 20,000 exact searches, one for each object in *A*. This requires ("only") 325,604,200 Euclidean distance calculations, but it requires approximately 5,676,400 disk accesses, for 1.04 days of wall clock time. This is clearly untenable.

We propose a simple algorithm to find the TSSD that exploits the fact that we can do both ultra-fast approximate search and fast exact search. We assume that set *B* is indexed and that set *A* is in main memory. The algorithm is sketched out in Table 7.

The algorithm begins by obtaining the approximate nearest neighbor in *B* for each time series in *A* (lines 5–10). A priority queue is created to order each time series in *A* according to the distance to its approximate nearest neighbor. Given that approximate

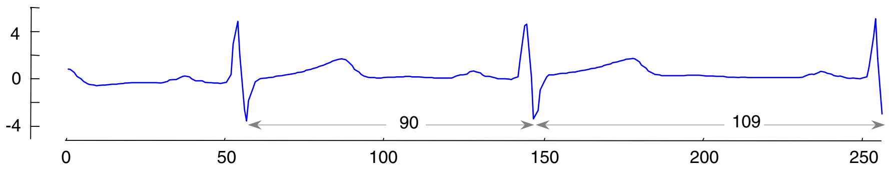

Fig. 19 The Time Series Set Difference discovered between ECGs recorded during a waking cycle and the previous 7.2 h

Table 7 An outline of an algorithm to find the TSSD

```
1  Function [IndexFile] = TimeSeriesSetDifference(A,B)
2  // sort priority queue by entry dist
3  PriorityQueue pq
4
5  foreach ts in A
6      IndexFile = B.ApproximateSearch(ts)
7      entry.dist = IndexFileDist(ts,IndexFile)
8      entry.ts = ts;
9      pq.Add(entry)
10 end
11
12 while !pq.IsEmpty
13     entry = pq.ExtractMax()
14     nextDist= pq.FindMax().dist
15
16     // exact search is suspended if the best-so-far
17     // becomes greater than nextDist
18     IndexFile = B.ExactSearch(entry,nextDist)
19
20     // IndexFile returns null when search is suspended
21     if IndexFile == null
22         pq.Add(entry)
23     else
24         return Indexfile
25     endif
26 end
```

search results are generally close to the exact answer, examining priority queue entries in order of descending distance is likely to be an effective heuristic in finding the entry which has the maximum nearest neighbor distance. The algorithm makes a minor addition to the exact search algorithm described previously. An additional parameter, nextDist, is required and denotes the distance value of the next entry at the top of the priority queue. If at any point during the exact search, the best-so-far falls below nextDist we suspend the search and return null. We can determine the state of exact search by checking the IndexFile for value (line 21). If the search was suspended, we reinsert the entry with its partially suspended state and updated distance back into the priority queue (line 22). Otherwise, if the IndexFile contains a search result, then we have obtained an exact answer whose nearest neighbor is larger than any other entry remaining in the priority queue, the TSSD (line 24).

To find the discordant heartbeats shown in Fig. 19, our algorithm did 43,779 disk accesses (20,000 in the first approximate stage, and the remainder during the refinement search phase), and performed 2,365,553 Euclidean distance calculations. The number of disk accesses for a sequential scan algorithm is somewhat better; it requires only 31,196 disk reads, about 71% of what our algorithm required. However, sequential scan requires 20,000,000,000 Euclidean distance calculations, which is 8,454 times greater than our approach and would require an estimated 6.25 days to complete. In contrast, our algorithm takes only 34 min.

Our algorithm is much faster because it exploits the fact that that most candidates in set *A* can be quickly eliminated by very fast approximate searches. In fact, of the 20,000 objects in set *A* for this experiment, only two of them (obviously including the eventual answer) had their true nearest neighbor calculated. Of the remainder, 17,772 were eliminated based only on the single disk access made in phase one of the algorithm, and 2,226 required more than one disk access, but less than a compete nearest neighbor search.

5.6 Batch nearest neighbor search

We consider another problem which can be exactly solved with a combination of approximate and exact search. The problem is that of batch nearest neighbor search. We begin with a concrete example of the problem before showing our *i*SAX-based solution. Here the context of DNA is used to provide a real world dataset with results which can be easily verified.

It has long been known that all the great apes except humans have 24 chromosomes. Humans, having 23, are quite literally the odd man out. This is widely accepted to be a result of an end-to-end fusion of two ancestral chromosomes. Suppose we do not know which of the ancestral chromosomes were involved in the fusion, we could attempt to efficiently discover this with *i*SAX.

We begin by converting DNA into time series. There are several ways to do this; here we use the simple approach shown in Table 8.

We converted Contig NT_005334.15 of the human chromosome 2 to time series in this manner, and then indexed all subsequences of length 1024 using a sliding window. There are a total of 11,246,491 base pairs (approximately 2,100 pages of DNA text written in this paper's format) and a total of 5,622,734 time series subsequences written to disk.

We converted 43 randomly chosen subsequences of length 1024 of chimpanzee's (Pan troglodytes) DNA in the same manner. We made sure that the 43 samples included at least one sample from each of the chimps 24 chromosomes.

We performed a search to find the chimp subsequence that had the nearest nearest-neighbor in the human reference set. Figure 20 shows the two subsequences plotted together. Note that while the original DNA strings are very similar, they are not identical.

Once again, this is a problem where a combination of exact and approximate search can be useful. To speed up the search we use *batch nearest neighbor search*. We define this as the search for the object O in a (relatively small) set *A*, which has the smallest nearest neighbor distance to an object in a larger set *B*. Note that to solve this problem, we really only need *one* exact distance, for object O, to be known. For the remaining

Table 8 An algorithm for converting DNA to time series

```
$T_1 = 0$;
For i = 1 to length(DNAstring)
    If $DNAstring_i$ = A,  then $T_{i+1} = T_i + 2$
    If $DNAstring_i$ = G,  then $T_{i+1} = T_i + 1$
    If $DNAstring_i$ = C,  then $T_{i+1} = T_i - 1$
    If $DNAstring_i$ = T,  then $T_{i+1} = T_i - 2$
End
```

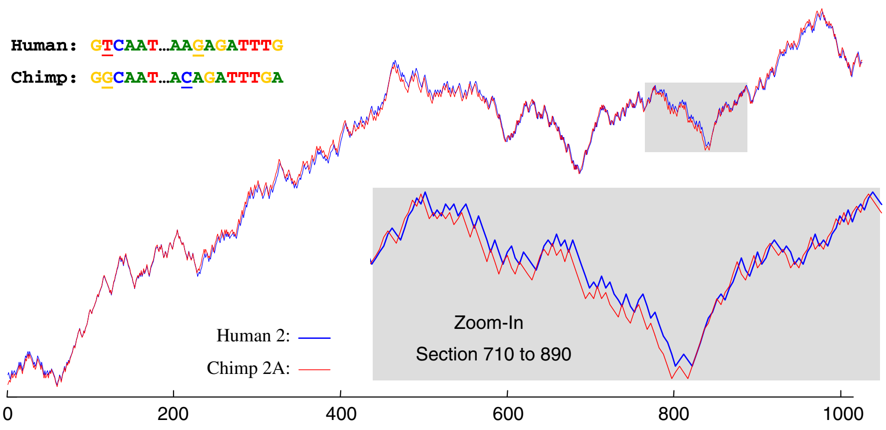

Fig. 20 Corresponding sections of human and chimpanzee DNA

Table 9 Batch nearest neighbor algorithm

```
1   Function [IndexFile] = BatchNearestNeighbor(A, B)
2   PriorityQueue pq
3
4   min.dist = inf
5   foreach ts in A
6       IndexFile = B.ApproximateSearch(ts)
7       dist = IndexFileDist(ts,IndexFile)
8       if min.dist > dist
9           min.dist = dist
10          min.ts = ts
11      endif
12
13      entry.dist = dist
14      entry.ts = ts;
15      pq.Add(entry)
16  end
17
18  pq.Remove(min.ts)
19  IndexFile = B.ExactSearch(min.ts)
20  min.dist = IndexFileDist(IndexFile)
21  min.IndexFile = IndexFile;
22
23  while !pq.IsEmpty
24      IndexFile = B.ExactSearch(pq.ExtractMin(), min.dist)
25      if IndexFile != null
26          min.dist = IndexFileDist(IndexFile)
27          min.IndexFile = IndexFile;
28      endif
29  end
30
31  return min.IndexFile;
```

objects in A, it suffices to know that a lower bound on their nearest neighbors is greater than the distance from O to its nearest neighbor. With this in mind, we can define an algorithm which is generally much faster than performing exact search for each of the objects in A. Table 9 outlines the algorithm.

The algorithm begins by obtaining the approximate search result for each time series in A (lines 5–16). Exact search is then performed on the query time series with the minimum approximate distance as an initial batch nearest neighbor candidate (line 19). We then perform exact search on the remaining time series in A using the current candidate as the initial distance to use during exact search (this requires a simple modification of lines 2–3 in the exact search algorithm detailed previously). By using the current batch nearest neighbor candidate as the initial distance value, we begin exact search immediately with a reduced distance value, increasing the likelihood of search space pruning from the very onset. If exact search returns a value, then a nearest neighbor less than the current best-so-far was found, and we update the best-so-far accordingly (lines 26–27). Once all time series are examined, the current best-so-far is returned as the batch nearest neighbor.

We can see this algorithm as an anytime algorithm (Xi et al. 2006; Zilberstein and Russell 1995). Recall that an anytime algorithm has the advantage that the quality of results increases monotonically with time and that execution is interruptible (after initial setup). Now considering the effect of diminishing returns and possible temporal constraints, it may be desirable to return an answer prior to the complete execution of the algorithm. An algorithm under the anytime framework facilitates this. For example, after the first phase, our algorithm has an approximate answer that we can examine. As the algorithm continues working in the background to confirm or adjust that answer, we can evaluate the current answer and make a determination of whether to terminate or allow the algorithm to persist.

In this particular experiment, the first phase of the algorithm returns an answer (which we later confirm to be the exact solution) in just 12.8 s, finding that the randomly chosen substring of chimp chromosome 2A, beginning at 7,582 of Contig NW_001231255 is a stunningly close match to the substring beginning at 999,645 of the Contig NT_005334.15 of human chromosome 2. The full algorithm terminates in 21.8 min. In contrast, a naive sequential scan takes 13.54 h.

5.7 Mapping the rhesus monkey chromosomes

In our final experiment we demonstrate the utility of ultra-fast approximate search by conducting a large scale indexing experiment on a real world dataset. In particular we will attempt to discover, and then align the rhesus macaque (*Macaca mulatta*) chromosomes that are homologous to the human chromosome 2 we encountered in the last section. Note that while the chimpanzee and human diverged only 6 million years ago, the human and macaque diverged 25 million years ago. We should therefore not expect that the matches will be as strikingly similar as the matches shown in Fig. 20. Instead, our approach will be to abandon any notion of *exact* searches, and conduct many *approximate* searches. We will then use the distributions of distances discovered for our approximate solutions to guide our hunt for homology, and to produce the alignment.

As we noted before, we are not claiming *i*SAX has a particular utility for such biological problems. It is merely a very large dataset for which we can obtain ground truth by other methods (Ijdo et al. 1991; Rogers et al. 2006).

To begin, we would like to determine some baseline which identifies the level of similarity we should expect between two chromosomes which are *not* related. This

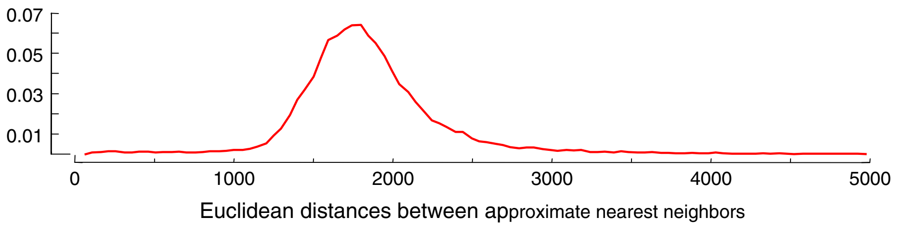

Fig. 21 The distribution of the Euclidean distances from subsequences in Rhesus Monkey chromosome 19 to their approximate nearest neighbor in Human chromosome 2. The distribution is normalized such that the area under the curve is one

could be done analytically, or with experiments on synthetic DNA. As we happen to known from external sources that the macaque chromosome 19 is unrelated to our target human chromosome 2, we will use that.

We converted human chromosome 2 to time series in the manner described in Sect. 5.6 and down sampled the time series by 4. Non-zero subsequences of length 1024 were extracted using a sliding window and indexed. From a total of 242,951,149 base pairs, 59,444,792 time series subsequences were indexed.

We then converted the macaque chromosome 19 to DNA time series using the same process and used each subsequence of 1024 as a query. Because DNA subsequences may have become been inverted some time in the last 25 million years (i.e. ..**TTGCAT**.. becomes ..**TCAGTT**..) we search for each time series, *and* its mirror image. In Fig. 21 we show the distribution of the Euclidean distance values from subsequences in macaque chromosome 19 and their approximate nearest neighbor in Human chromosome 2.

This distribution tells us that the average distance between nearest neighbors is approximately 1,774 with a standard deviation of 301. There are very few distances less than 1,000 or greater than 4,000. If we repeat the experiment with randomly generated data or other non-related DNA sequences we find nearly identical distributions is every case. We therefore can use this distribution as a baseline for a systematic search through the remaining 19 macaque chromosomes. While we could use a statistical test such as the Kullback–Leibler divergence (Fuglede and Topsøe 2004), we simply visually inspected the distributions. Two of the monkey chromosomes, 12 and 13, produce significantly different distributions. In Fig. 22 we show a comparison of the distributions for macaque chromosome 19 and 12.

Because both chromosome 12 and 13 from the macaque have a suspicious divergence from the expected distribution, we can create a dot plot to see which sequences in the monkey, map closely to which sequences in the human. We need to set some threshold, because we are not interested in knowing where the nearest neighbor to a subsequence is, if that nearest neighbor happens to be relatively far away. We observe from Fig. 23 that the two distributions start to diverge (reading right to left) at about 1,250, so we use that value as the cutoff distance for dot plot construction.

This figure suggests that essentially all of human chromosome 2 can be explained by a fusion of Rhesus Monkey chromosome 12 and 13 (or vice versa). Of course, it has been suspected that that human chromosome 2 is a recent species-specific fusion of

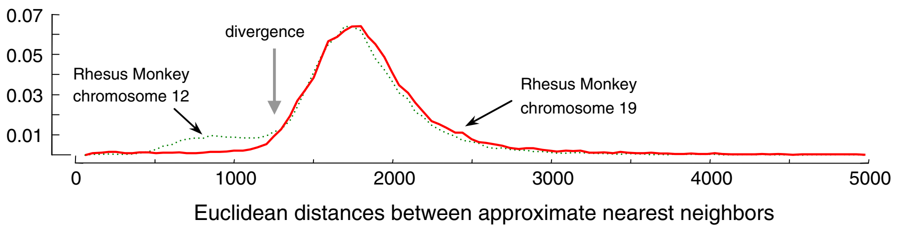

Fig. 22 The distribution of the Euclidean distances from subsequences in Rhesus Monkey chromosomes 19 and 12, to their approximate nearest neighbor in Human chromosome 2

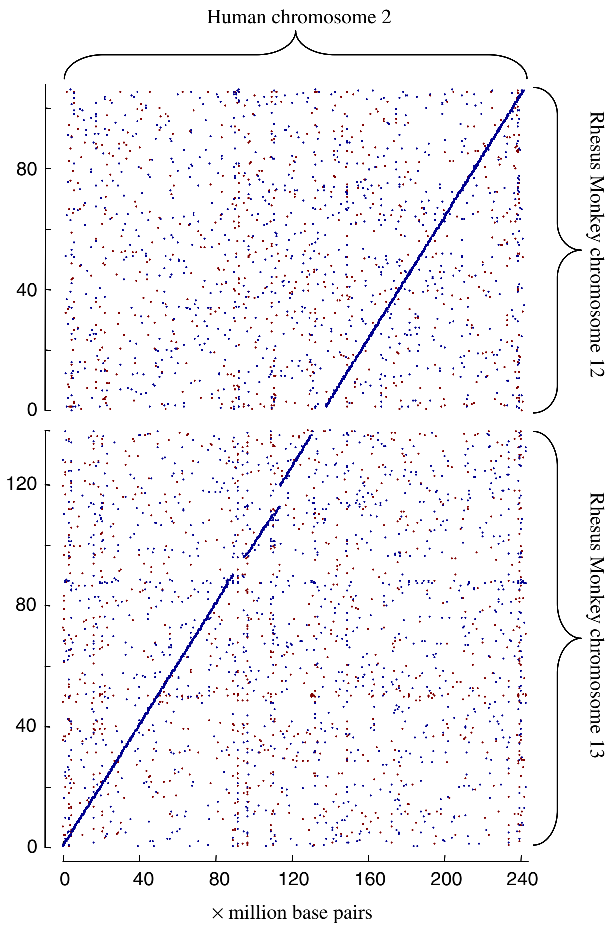

Fig. 23 A dot plot showing the alignment of Human chromosome 2 with both chromosome 12 and 13 of the Rhesus Monkey. Each dot represents a location where a subsequence in the monkey (row) is less than 1,250 from a subsequence in a human (column)

two ancestral primate chromosomes for several decades (Ijdo et al. 1991). More recent studies (Rogers et al. 2006) have confirmed that the mapping above is correct, and the section that the rhesus macaque 12 and 13 maps to are called 2q and 2p, respectively.

In total, we performed 119,400 approximate queries taking slightly little over 4 h, whereas a naive method of scanning subsequences of one chromosome across the other would result in nearly 119,400 * 59,444,792 distance computations, requiring well over a year, a duration which is clearly unacceptable for such applications.

6 Conclusions

We introduced *i*SAX, a representation that supports indexing of massive datasets, and have shown it can index up to one hundred million time series. We have also provided examples of algorithms that use a combination of approximate and exact search to ultimately produce exact results on massive datasets. Other time series data mining algorithms such as motif discovery, density estimation, anomaly discovery, joins, and clustering can similarly take advantage of combining both types of search, especially when the data is disk resident. We plan to consider such problems in future work.

References

André-Jönsson H, Badal DZ (1997) Using signature files for querying time-series data. In: Proceedings of the 1st PKDD, pp 211–220

Assent I, Krieger R, Afschari F, Seidl T (2008) The TS-Tree: efficient time series search and retrieval. In: Proceedings of the 11th EDBT

Bagnall AJ, Ratanamahatan C, Keogh E, Lonardi S, Janacek GJ (2006) A Bit Level Representation for time series data mining with shape based similarity. Data Min Knowl Disc 13(1):11–40

Batista LV, Melcher EUK, Carvalho LC (2001) Compression of ECG signals by optimized quantization of discrete cosine transform coefficients. Med Eng Phys 23(2):127–134

Bingham E, Mannila H (2001) Random projection in dimensionality reduction: applications to image and text data. In: Proceedings of the seventh ACM SIGKDD international conference on knowledge discovery and data mining, San Francisco, California, August 26–29, 2001. KDD '01, ACM, New York, NY, pp 245–250

Cai Y, Ng R (2004) Indexing spatio-temporal trajectories with Chebyshev polynomials. In: Proceedings of the ACM SIGMOD, pp 599–610

Chan K, Fu AW (1999) Efficient time series matching by wavelets. In: Proceedings of 15th international conference on data engineering, pp 126–133

Chen J, Itoh S (1998) A wavelet transform-based ECG compression method guaranteeing desired signal quality. IEEE Trans Biomed Eng 45(12):1414–1419. doi:10.1109/10.730435

Chen Q, Chen L, Lian X, Liu Y, Yu JX (2007) Indexable PLA for efficient similarity search. In: Proceedings of the 33rd international conference on very large data bases

Ding H, Trajcevski G, Scheuermann P, Wang X, Keogh E (2008) Querying and mining of time series data: experimental comparison of representations and distance measures. In: Proceedings of the VLDB endow, 1, 2 (Aug 2008), pp 1542–1552

Faloutsos C, Ranganathan M, Manolopoulos Y (1994) Fast subsequence matching in time-series databases. In: Proceedings of the ACM SIGMOD

Fuglede B, Topsøe F (2004) Jensen-Shannon divergence and hilbert space embedding. In: Proceedings of the international symposium on information theory

Guttman A (1984) R-trees: a dynamic index structure for spatial searching. SIGMOD Rec 14(2):47–57. doi:10.1145/971697.602266

Huang Y, Yu PS (1999) Adaptive query processing for time-series data. In: Proceedings of the 5th ACM SIGKDD, pp 282–286

Ijdo J, Baldini A, Ward DC, Reeders ST, Wells RA (1991) Origin of human chromosome 2: an ancestral telomere–telomere fusion. Proc Natl Acad Sci USA 88:9051–9055. doi:10.1073/pnas.88.20.9051

Kaffka S, Wintermantel B, Burk M, Peterson G (2000) Protecting high-yielding sugarbeet varieties from loss to curly top. http://sugarbeet.ucdavis.edu/Notes/Nov00a.htm

Keogh E (2008) www.cs.ucr.edu/~eamonn/SAX.htm

Keogh E, Shieh J (2008) *i*SAX home page. www.cs.ucr.edu/~eamonn/iSAX/iSAX.htm

Keogh E, Chakrabarti K, Pazzani MJ, Mehrotra S (2001a) Dimensionality reduction for fast similarity search in large time series databases. KAIS 3(3):263–286. doi:10.1007/PL00011669

Keogh E, Chakrabarti K, Pazzani M, Mehrotra S (2001b) Locally adaptive dimensionality reduction for indexing large time series databases. In: Proceedings of ACM SIGMOD conference on management of data, May, pp 151–162

Kumar N, Lolla N, Keogh E, Lonardi S, Ratanamahatana CA, Wei L (2005) Time-series bitmaps: a practical visualization tool for working with large time series databases. In: Proceedings of SIAM international conference on data mining

Lin J, Keogh E, Wei L, Lonardi S (2007) Experiencing SAX: a novel symbolic representation of time series. Data Min Knowl Disc 15:107–144

Megalooikonomou V, Wang Q, Li G, Faloutsos C (2005) A multiresolution symbolic representation of time series. In: Proceedings of the 21st ICDE

Morinaka Y, Yoshikawa M, Amagasa T, Uemura S (2001) The L-index: an indexing structure for efficient subsequence matching in time sequence databases. In: Proceedings of Pacific-Asian conference on knowledge discovery and data mining

Portet F, Reiter E, Hunter J, Sripada S (2007) Automatic generation of textual summaries from neonatal intensive care data. In: Proceedings of AIME 2007

Ratanamahatana CA, Keogh E (2005) Three myths about dynamic time warping. In: Proceedings of SIAM international conference on data mining (SDM '05), pp 506–510

Rogers J et al (2006) An initial genetic linkage map of the rhesus macaque (Macaca mulatta) genome using human microsatellite loci. Genomics 87(1):30–38. doi:10.1016/j.ygeno.2005.10.004

Scholle S, Schäfer T (1999) Atlas of states of sleep and wakefulness in infants and children. Somnologie - Schlafforschung und Schlafmedizin 3(4):163

Shatkay H, Zdonik SB (1996) Approximate queries and representations for large data sequences. In: Su SY (ed) Proceedings of the 12th international conference on data engineering, ICDE, IEEE Computer Society, Washington, DC, February 26–March 01, 1996, pp 536–545

Steinbach M, Tan P, Kumar V, Klooster S, Potter C (2003) Discovery of climate indices using clustering. In: Proceedings of the ninth ACM SIGKDD, pp 446–455

Wei L, Keogh E, Van Herle H, Mafra-Neto A (2005) Atomic wedgie: efficient query filtering for streaming times series. In: Proceedings of the fifth IEEE international conference on data mining, pp 490–497

Xi X, Keogh E, Shelton C, Wei L, Ratanamahatana CA (2006) Fast time series classification using numerosity reduction. In: Proceedings of the 23rd ICML, pp 1033–1040

Zilberstein S, Russell S (1995) Approximate reasoning using anytime algorithms. In: Imprecise and approximate computation. Kluwer Academic Publishers