Group SAX: Extending the Notion of Contrast Sets to Time Series and Multimedia Data

Jessica Lin[1] and Eamonn Keogh[2]

[1] Information and Software Engineering
George Mason University
jessica@ise.gmu.edu
[2] Department of Computer Science & Engineering
University of California, Riverside
eamonn@cs.ucr.edu

Abstract. In this work, we take the traditional notation of *contrast sets* and extend them to other data types, in particular time series and by extension, images. In the traditional sense, contrast-set mining identifies attributes, values and instances that differ significantly across groups, and helps user understand the differences between groups of data. We reformulate the notion of contrast-sets for time series data, and define it to be the key pattern(s) that are maximally different from the other set of data. We propose a fast and exact algorithm to find the contrast sets, and demonstrate its utility in several diverse domains, ranging from industrial to anthropology. We show that our algorithm achieves 3 orders of magnitude speedup from the brute-force algorithm, while producing exact solutions.

1 Introduction

As noted by Bay and Pazzani, "*A fundamental task in data analysis is understanding the differences between several contrasting groups*". While there has been much work on this topic for discrete and numeric objects, to the best of our knowledge, the problem of mining contrast sets for time series or other multimedia objects has not been addressed before. This work makes two fundamental contributions to the problem.

- We introduce a formal notion of time series contrast set.
- We introduce a fast and exact algorithm to find time series contrast sets.

Contrast-set mining is a relatively new data-mining task, designed to identify differences among various groups of data. It can be roughly viewed as a variant of association rule mining [1, 4, 9]. While association rule mining discovers rules that describe or explain the current situation, contrast-set mining finds rules that differentiate contrasting groups of data, by identifying attributes and values (or conjunctions thereof) that differ meaningfully across them [2, 3]. Knowing these attributes that characterize the discrepancies across various groups can help users understand the fundamental differences among them, and make independent decisions on those groups accordingly.

J. Fürnkranz, T. Scheffer, and M. Spiliopoulou (Eds.): PKDD 2006, LNAI 4213, pp. 284 – 296, 2006.

In this work, we introduce the concept of contrast-set mining for time series datasets. Due to the unique characteristics of time series data, the notion of contrast-set mining deviates from the traditional sense as defined on discrete data. More specifically, time series contrast-set mining aims to identify the key patterns rather than rules that differentiate two sets of data.

While this paper addresses *time series* contrast set explicitly, we note it is possible to convert many kinds of data into time series. For example, Figure 1 shows that we can convert shapes to time series. Other types of data, from text to video to XML [11], have also been converted into time series with varying degrees of success.

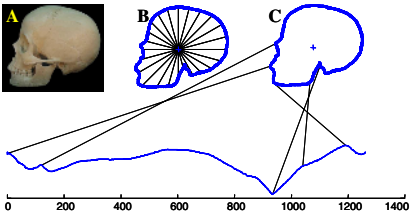

Fig. 1. Shapes can be converted to time series. A) A bitmap of a human skull. B) The distance from every point on the profile to the center is measured and treated as the Y-axis of a time series of length *n* (C) The mapping of the skull to time series.

Time series contrast-set mining should not be confused with clustering or summarization, in which the aim is to compare the *average* behavior of different sets. Suppose we are given two datasets that come from the same source; for example, two sets of time series telemetry from two shuttle launches, or two sets of ECG heartbeats from one patient – one before a drug was given and one after, or two sets of skulls, one set from Asia and one set from Europe. For some external reason, we suspect or actually know that there is something different about one group - maybe one shuttle crashed, or the patient had a heart attack. It might be the case that one entire group is different and this explains the problem. Maybe *all* the shuttle telemetry from the first launch has an amplitude scaling increase, or maybe *all* the ECG collected after the drug was administered show a faster heart rate. However, maybe a single or a few object(s) explain the difference. This is *exactly* what we are aiming to find.

As we shall show, our work has potential applications in detecting anomalies or differences in time series datasets in several diverse domains, and in certain images. The paper is organized as follows. In Section 2 we provide a brief overview on related work and necessary background. Section 3 describes the brute-force algorithm on finding time series contrast sets. In Section 4, we introduce an algorithm that offers 3 orders of magnitude speedup from the brute-force algorithm. Section 5 shows some experimental results on some time series datasets and image data. Section 6 concludes and offers suggestion for future work.

2 Related Work and Background

The most representative work on contrast-set mining is perhaps the work by Bay and Pazanni [2]. The authors introduce the task of contrast-set mining, and their algorithm STUCCO offers efficient search through the space of contrast-sets, based on the Max-Miner [4, 14] rule-discovery algorithm. Follow-up work by Webb et al. [14] discovered that existing commercial rule-finding system, Magnum Opus [13], can successfully perform the contrast-set task. The authors conclude that contrast-set mining is a special case of the more general rule-discovery task. He et al. [6] applies contrast-set mining on customer clusters to find cluster-defined actionable rules. In [10] Minaei-Bidgoli et al. proposed an algorithm to mine contrast rules on a web-based educational system. Their work is unique in that it allows rule discovery with very low minimum support, so rules that would have been otherwise overlooked can be discovered.

2.1 Notation

For simplicity and clarity we will speak only of time series below; however, as we hinted in Figure 1, and as we explicitly address in Section 5.2, these ideas can be extended to images and other types of data

Definition 1. *Time Series*: A time series $T = t_1,\ldots,t_m$ is an ordered set of m real-valued variables.

For data mining purposes, we are typically not interested in any of the *global* properties of a time series; rather, we are interested in *local* subsections of the time series, which are called subsequences.

Definition 2. *Subsequence*: Given a time series T of length m, a subsequence C of T is a sampling of length $n \leq m$ of contiguous position from p, that is, $C = t_p,\ldots,t_{p+n-1}$ for $1 \leq p \leq m - n + 1$.

Since all subsequences may potentially be attributing to contrast sets, any algorithm will eventually have to extract all of them; this can be achieved by use of a sliding window. We also need to define some distance measure $Dist(C,M)$ in order to determine the similarity between subsequences. We require that the function $Dist$ be symmetric, that is, $Dist(C,M) = Dist(M,C)$. There are dozens of distance measures for time series in the literature. However recent evidence strongly suggests that simple Euclidean distance is very difficult to beat for classification and clustering problems, so without loss of generality we will exclusively use it in this work.

Definition 3. *Euclidean Distance*: Given two time series Q and C of length n, the Euclidean distance between them is defined as:

$$Dist(Q,C) \equiv \sqrt{\sum_{i=1}^{n}(q_i - c_i)^2}$$

Each time series subsequence is normalized to have mean zero and a standard deviation of one before calling the distance function, because it is well understood that in virtually all settings, it is meaningless to compare time series with different offsets and amplitudes [7].

We may wish to be able to find a pattern that occurs in one time series, but does not have a close match in another. Since such patterns could be used to differentiate time series, we call such patterns TS-Diffs.

Definition 4. *TS-Diff*: Given two time series T and S, and a user-given parameter n, *TS-Diff*(T,S,n) is the subsequence C of T, of length n, that has the largest distance to its closest match in S.

Note that this definition is not generally symmetric; that is, *TS-Diff*$(T,S,n) \neq$ *TS-Diff*(S,T,n). We may wish to be able to find the patterns that differentiate the two time series.

Definition 5. *Time Series Contrast Sets*: Given two time series T and S, a user given parameter n, let C = *TS-Diff(T, S, n)* and D = *TS-Diff(S, T, n)*. The contrast set CS for T and S is $\{C, D\}$.

Often there might be more than one pattern that differentiates two time series. The definition above can be easily extended to K time series contrast sets.

3 Finding Time Series Contrast Sets

We begin by describing the simple and obvious brute-force algorithm for finding the contrast sets between two time series T and S. For simplicity, let's assume that the two sets of data are of the same length, m, although in reality, their lengths need not be the same. Furthermore, let's assume that we are interested in finding patterns or subsequences of length n that differentiate the data. Since the results are not necessarily symmetric, we need to process T and S separately.

Intuitively, the definition of contrast sets tells us that the brute-force algorithm will need to compute the pairwise distances between all subsequences in T and S. That is, for each subsequence C in T, we need to compute its distance to all subsequences in S, in order to determine the distance to its closest match in S. The subsequence in T that has the greatest such value is then *TS-Diff (T, S, n)*. This can be achieved with nested loops, where the outer loop considers each candidate subsequence in T, and the inner loop is a linear scan to identify the candidate's nearest match in S. The brute-force algorithm is easy to implement and produces exact results. However, its $O(m^2)$ time complexity makes this approach untenable for even moderately large datasets.

Fortunately, the following observations offer hope for improving the algorithm's running time. Recall what we wish to find here: we wish to identify the subsequence in T that is farther away from its nearest match in S than all other subsequences are from their respective nearest matches. Hence, we can keep track of the candidate that has the largest nearest match distance so far. This implies that we might not need to know the actual nearest match distance for every subsequence in T. The only piece of information that is crucial is whether or not a given subsequence in T can be the candidate for *TS-Diff*. This can be achieved by determining if the given subsequence has the *potential* of having a large nearest match distance.

Suppose we use the variable best_so_far_dist to keep track of the largest nearest-match distance so far. Consider the following scenario. Suppose after examining the first subsequence in T, we find that this subsequence is 10 units away from its nearest match in S (and we initialize best_so_far_dist to be 10). Now suppose as we move on to the next candidate subsequence in T, we find that it's 2 units away from the first

subsequence in *S*. At this point, we know that the current candidate could not be the candidate for *TS-Diff*, since its nearest match distance is *at most* 2 units, which is far less than best_so_far_dist. We can therefore safely abandon the rest of the search for this candidate's nearest match. In other words, when we consider a candidate subsequence *C* in *T*, we don't actually need to find its true nearest match in *S*. As soon as we find any subsequence in *S* that has a smaller distance to *C* than best_so_far_dist, we can abandon the search process for *C*, safe in the knowledge that it could not be *TS-Diff*.

Clearly, the utility of such optimization depends on the order in which the subsequences in *T* are considered, as well as the order in which the subsequences in *S* are matched against the current candidate. The earlier we examine a subsequence in *S* that has a smaller Dist(C, D) than best_so_far_dist, the earlier we can abandon the search process for the current candidate. This observation brings us to reducing the *TS-Diff* problem into a generic framework where, instead of examining all subsequences in sequential order, it allows one to specify any customized ordering of the subsequences to be examined. Table 1 shows the pseudocode.

Table 1. Heuristic TS-Diff Discovery

```
1   Function [dist, I]= Heuristic_Search(T, S, n, Outer, Inner)
2    best_so_far_dist = 0
3    best_so_far_TS = NaN
4
5   // Begin Outer Loop
6   For Each C in T ordered by heuristic Outer
7      nearest_neighbor_dist = infinity
8
9      // Begin Inner Loop
10     For Each D in S ordered by heuristic Inner
11        IF Dist(C, D) < best_so_far_dist
12           Break                    // Break out of Inner Loop
13        End
14        IF Dist(C, D) < nearest_neighbor_dist
15           nearest_neighbor_dist = Dist(C, D)
16        End
17     End                           // End Inner Loop
18     IF nearest_neighbor_dist > best_so_far_dist
19        best_so_far_dist = nearest_neighbor_dist
20        best_so_far_TS  = C
21     End
22  End                              // End Outer Loop
23  Return[ best_so_far_dist, best_so_far_TS]
```

We can consider the following to be the best scenario: for the ordering *Outer*, the subsequences in *T* are sorted by descending order of distances to their closest matches in *S*, so that the true *TS-Diff* is the first object examined, and best_so_far_dist is at its maximum value after the first iteration of the outer loop. For the ordering *Inner*, the subsequences in *S* are sorted in ascending order of distances to the current candidate *C* so that it's guaranteed that the first object examined in the inner loop will have a distance smaller than best_so_far_dist (otherwise *C* would have been placed towards the front of the queue). For this heuristic, the first invocation of the inner loop will run to completion to determine best_so_far_dist. Thereafter, all subsequent invocations of the inner loop will be abandoned after only one iteration, i.e. after discovering that the current distance is

smaller than best_so_far_dist. The total time complexity is thus O(m). This is the best possible scenario. We call this heuristic *magic*.

On the other hand, we should expect the worst-case scenario to be the exact opposite of the best-case scenario. In this scenario, the true *TS-Diff* will be the last object to be examined. More specifically, this heuristics has the worst-possible ordering such that for *Outer*, the subsequences in *T* are sorted by ascending order of distance to the closest match in *S*. For *Inner*, the time series in *S* are sorted in descending order of distance to the current candidate. In this case, we are back to the quadratic time complexity as the brute-force algorithm. This is the *perverse* heuristic.

Another possible strategy is to order the subsequences randomly for both *Outer* and *Inner* heuristics. Empirically it works reasonably well, and the inner loop is usually abandoned early, considerably speeding up the algorithm.

The three strategies discussed so far suggest that a linear-time algorithm is possible, but only with the aid of some very wishful thinking. The best-case heuristic requires perfect orderings of subsequences in the inner and outer loops. The only known way to produce such ordering is to actually compute the distances, which indirectly solve the problem. Even if the distances are known in advance, any sorting algorithm requires at least O(*mlogm*) time complexity. To ensure that the total running time is no worse than the worst-case running time, we must require a linear-time heuristic for *T* (outer loop, invoked once for the whole program), and a constant-time heuristic for every invocation of S ordering.

Observe that, however, for *Outer*, we do not actually need to achieve a perfect ordering to achieve dramatic speedup. All we really require is that among the first few candidate subsequences being examined, we have at least one that has a large distance to its closest match. This will give the best_so_far_dist variable a large value early on, which will allow more early terminations of the inner loop.

Similar observation goes for *Inner*. In the inner loop, we also do not actually need a perfect ordering to achieve dramatic speedup. We just need that among the first few subsequences in *S* being examined, we have at least one that has a distance to the current candidate that is smaller than the current value of best_so_far_dist. This is a sufficient condition to allow early termination of the inner loop.

We can imagine a full spectrum of algorithms, which only differ by how well they order subsequences relative to the best-case ordering. The random heuristic is somewhere between the best and the worst heuristics. Our goal then is to find the best possible approximations to the best-case heuristic ordering, which is the topic of the next section.

4 Group SAX: Approximating the Best-Case Heuristic

Our techniques for approximating the perfect ordering returned by the hypothetical best-case heuristic require us to discretize the real-valued time series data first. We choose **S**ymbolic **A**ggregate Appro**X**imation (SAX) representation of time series introduced in [8] to be the discretization technique. Since our algorithm finds differences between groups of time series using SAX, we call it *Group SAX*.

SAX works by first approximating the original time series of length *m* with *w* coefficients ($w << n$) via Piecewise Aggregate Approximation (PAA) [8]. These *w* coefficients are then converted to symbols of cardinality α, according to where they

reside in the Gaussian space. Therefore, the discrete approximation of the time series is a string of length *w*. Due to the space constraints, we direct interested readers to [8] for more information on SAX.

4.1 The *Outer* Loop Heuristic

We begin by creating two data structures to support our heuristics. Before that, there are two parameters associated with SAX that we must consider. They are the cardinality of the SAX alphabet size α, and the SAX word size *w*. Extensive experiments carried out by the current authors and dozens of other researchers worldwide [8] suggest that a value of either 3 or 4 for α is best for virtually any task on any dataset. Furthermore, while the choice of *w* depends on the data, we observe empirically that the speedup does not critically depend on *w*. We refer interested readers to [8] for more details on SAX parameter setting, but note that the parameters only affect the efficiency of the algorithm, not the final results. The subsequences are extracted by sliding a window of length n across the time series, which are then converted to SAX words. These SAX words are inserted to an array where the index refers back to the original sequence. Figure 2 gives a visual intuition of this, where both α and *w* are set to 3. Once we have this ordered list of SAX words, we construct a hash table. Each bucket in the hash table represents a unique word and contains a linked-list index of all subsequences that map to the corresponding string. The hash function we used assigns each SAX word a unique address ranging from 0 to $\alpha^w - 1$, hence guarantees minimal perfect hashing. The memory consumption for creating an empty hash table is considerably small and can be ignored.

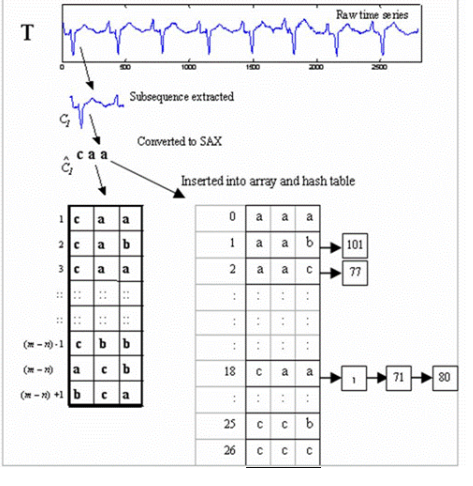

Fig. 2. The data structure used to support the *Inner* and *Outer* heuristics. (*left*) An array of SAX words (*right*). An excerpt of the hash table that contain pointers to the corresponding subsequences.

The process is repeated for both time series datasets that we wish to contrast. Each dataset has its own array and hash table. Once the hash tables are constructed for *T* and *S*, we can now determine the ordering of the outer loop objects. Recall that for *Outer*, the goal is to have a large nearest-match distance early in the loop. Thus, intuitively, we want to identify the subsequences in *T* that have the smallest count of matching SAX word in *S* (virtually zero). The reason is simple. Since SAX approximates and captures similarities between time series, we can expect that similar time series are likely to map to the same SAX representation. Conversely, subsequences mapped to SAX strings that are exclusive to *T*, i.e., they appear in *T* but not in *S,* are unlikely to have close matches in *S*. Therefore, by considering the candidate subsequences that map to such unique or rare SAX words with respect to S early in the outer loop, we have a great chance of obtaining large best_so_far_dist value among the first few candidates examined, thus allowing early termination in the subsequent inner-loop iterations.

To achieve this, we utilize the two hash tables we build from *T* and *S*. We perform a linear scan in *Hash_S*, and for each empty bucket we encounter, check if the corresponding bucket in *Hash_T* is empty as well. If not, then we record the subsequence indices from this bucket in *Hash_T*, and add them to a list which we call *Preferred_List*. We end up with a list that contains indices of subsequences whose SAX representations are unique in *T*. The subsequences referred to by the *Preferred_List* will be given to the outer loop to search over first. After the outer loop has exhausted this set, the rest of the candidates are visited in random order.

4.2 The *Inner* Loop Heuristic

Our *Inner* heuristic also leverages off *Hash_T* and *Hash_S*. Recall that in the inner loop, as soon as we encounter a subsequence similar enough to the current candidate in the outer loop, such that their distance is smaller than best_so_far_dist, then the search for the current candidate can be abandoned. The heuristic used for the outer loop gives us hope that best_so_far_dist will take on a large value early on in the process. For the inner loop, we take the optimization one step further by putting the subsequences that might cause early termination in the front of the queue so they are examined first. By identifying and eliminating those that could not possibly have a nearest match distance larger than best_so_far_dist early in the iteration, many unnecessary computations can be spared. Note that all it takes is having one distance that is smaller than best_so_far_dist.

To achieve this, we determine the ordering of the inner loop as follows. When candidate *i* is first considered in the outer loop, we look up the SAX word that it maps to, by examining the i^{th} word in the array for *T*. We then compute the key for the SAX word, and order the first items in the inner loop in the order of the elements in the linked list index found at the corresponding bucket in *Hash_S*. These subsequences will be visited first by the inner loop. After this list is exhausted, the rest of the subsequences are visited in random order.

Note that in the beginning, since the outer loop considers the unique words in *T* first, there will be no matching words in S, thus no optimization for the inner loop. One option would be to limit the size of *Preferred_List* in the outer loop; however, empirically we find that even without doing so, the speed up is already significant that it is not necessary to put a threshold on the number of “unique” subsequences to examine first in the outer loop.

5 Empirical Evaluation

5.1 The Utility of Contrast Sets: Time Series

We begin by considering the contrast set between German and Italian consumer electrical power demands. We obtained the last 9 nine years data from two towns, Dortmumd and Florence. Both time series are sampled hourly, and thus each is of length 269,379. We are interested in monthly patterns, so we set parameter *n* to 672 (4 weeks). Figure 3 shows the number one TS-Diff(*Italian*, *German*, 672) found.

The difference is striking. The German time series simply shows the typical weekly pattern repeated 4 times. The typical weekly pattern for power demand corresponds to 5 major peaks for the Monday to Friday 9 to 5 work hours, followed by a much smaller peak for Saturday and minimal power demand for Sunday. In contrast, the Italian demand shows a dramatic reduction in power demand as the month of August progresses. What is behind this difference?

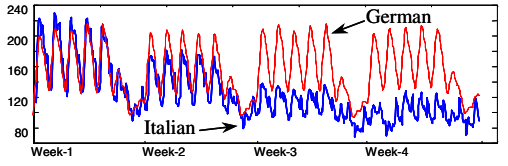

Fig. 3. The four weeks of Italian Power Demand, beginning on Monday July 31st 1995 is radically different from the *most* similar four weeks of German Power Demand, beginning on 3rd of May 1999.

The answer lies in an Italian cultural phenomenon. According to travel writer Nella Nencini, "By the middle of July, normal activity begins to wane and by the beginning of August, shops no longer close between 1 and 4 p.m., they close for two or three weeks. Dry cleaners close, mechanics close, factories close, wineries close, restaurants close, even some museums close. Cities like Florence and Venice would be abandoned if not for the tourists braving the heat to visit artistic treasures". The dramatic change in power demand reflects the fact that most major employers (like Fiat and many government offices) simple shut down for the month. This difference is obvious if we zoom out and look at a full year of Florence's power demand, as shown in Figure 4.

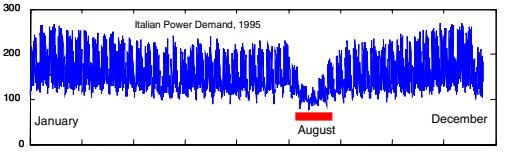

Fig. 4. One Year of Italian Power Demand (1995). Note that August is radically different from the rest of the year.

5.2 The Utility of Contrast Sets: Shapes

As noted in Section 1, we can trivially apply our ideas to shapes, since shapes typically represented as a 1-dimensional signal. In this section we show some examples of mining image datasets.

Petroglyphs are images incised in rock by prehistoric peoples. They were an important form of pre-writing symbols, used in communication from approximately 10,000 B.C. to modern times, depending on culture and location. Petroglyphs have been found on every continent except Antarctica. In virtually all cases, there is still great controversy about who created the petroglyphs, when, and for what purpose. The controversy is not for the want of evidence, for example the petroglyph shown in Figure 6A is just one of more than 100,000 petroglyphs to be found in an area of a mere 91 square miles of Sierra Nevada called Renegade Canyon.

We believe that contrast sets offer one possible tool for examining massive archives of petroglyph images. As a preliminary experiment we began by contrasting petroglyphs from the aforementioned site in Nevada with a similarly dense site in Sheep Springs in Kern County California.

For simplicity we only consider petroglyphs of animals, however it has be estimated that 51% of the Renegade Canyon petroglyphs are of animals, and as the name "Sheep Springs" suggests, the Californian site is similarly dense with images of sheep.

There are several technical problems that must be faced before we apply our algorithm. First of all there is the issue of image processing and shape extraction. This task is non-trivial, but not of direct interest here. Note that the representation we use is scale and translation invariant.

Once the shapes have been extracted and converted to time series we must consider two important issues. Do we wish to be enantiomorphic invariant? That is to say do we wish to attach any significance to whether an animal is facing left or right? After consulting with an anthropologist we decided to ignore such directional information. This we achieve by simply augmenting the database of time series to include the mirror image of each image. We can achieve this directly in the time series representation. Recall that a single time series C is defined as: $C = c_1,c_2,\ldots,c_j,\ldots,c_n$.

For each such time series we also add C' to the database: $C' = c_n,c_{n-1},\ldots,c_2,\ldots,c_1$.

We must also consider the problem of rotation invariance. Should we attach any significance to the angle at which the animals are drawn? For two reasons our answer is no. The first is pragmatic. Finding "correct" orientation of a shape is difficult problem in general. The second reason why we choose to be invariant to orientation is an observation by our anthropologist that often the animals are drawn to align with cracks and fissures in the rocks, and the orientation appears to have no significance.

Once again, achieving rotation invariance in our representation is easy to achieve by augmenting our database to contain additional time series. Let $\mathbf{C}$ be all n circular shifts of C:

$$\mathbf{C} = \begin{Bmatrix} c_1, c_2, \ldots, c_{n-1}, c_n \\ c_2, \ldots, c_{n-1}, c_n, c_1 \\ \vdots \\ c_n, c_1, c_2, \ldots, c_{n-1} \end{Bmatrix}$$

By adding all such circular shifts to our database we achieve rotation invariance.

Figure 5 shows some examples of shapes from Renegade Canyon and their closest match from Sheep Springs, including the one that maximizes TS_Diff(Renegade Canyon, Sheep Springs).

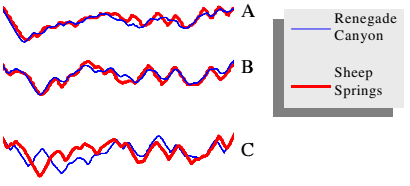

Fig. 5. Three petroglyphs shapes from Renegade Canyon (light lines) and their best matches from the Sheep Springs database. While most shapes are like A and B in having a close match in the other database, the shape from Renegade Canyon shown in C is unusually different from its nearest counterpart in Sheep Springs.

Why is one image from Renegade Canyon so different to any image from Sheep Springs? An inspection of the original images, as shown in Figure 6, reveals the answer. While there are a handful of images that show a spear stuck into the body of a sheep in Renegade Canyon, including the one shown in Figure 6 (top), a careful manual inspection of the Sheep Springs database reveals that there are *no* such petroglyphs in Sheep Springs.

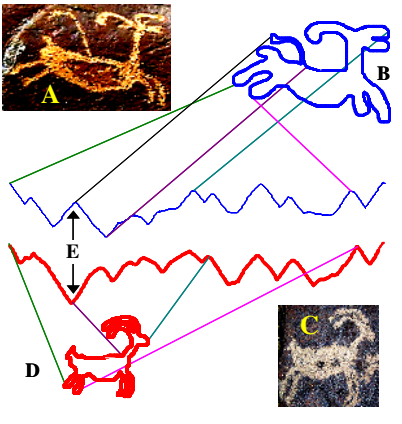

Fig. 6. A) A petroglyph from Renegade Canyon of a Bighorn Sheep with a spear stuck in it has its profile automatically extracted (B) and converted to a time series. C) A petroglyph from Sheep Springs of a Bighorn Sheep with a spear stuck in it has it profile automatically extracted (D) and converted to a time series. Note that the two shapes are not a good match, mainly because the part of the time series corresponding to the spear (E) creates a relatively large Euclidean distance between the two shapes.

5.3 The Utility of Our Search Technique

In Figure 7, we compare the brute force algorithm to our algorithm in terms of the number of times the Euclidean distance function is called. Since we are now

interested in the scalability of both approaches, we use random walk datasets of lengths 100,000 and 200,000. For our algorithm we averaged the results over 100 runs for each data length.

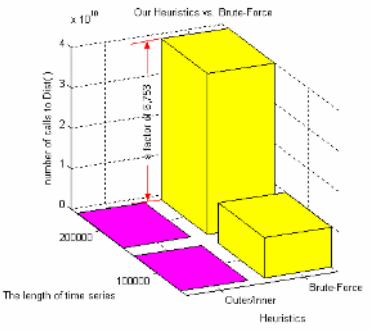

Fig. 7. The number of calls to the distance function required by brute force and heuristic search. The window length is 64 for all cases.

Note that as the data sizes increase, the differences get larger. For a time series of length 200,000, our approach is almost seven thousand times faster than brute force approach. This experiment is in fact pessimistic in that we used the datasets (random walk) that did not have any obvious structures in them. In general, if the data exhibit some structures, as the ones used in Section 5.1 and 5.2, then our approach would be even faster since there would be a lot more potential matches for most subsequences, to allow early termination.

6 Conclusions and Future Work

In this work, we have introduced the notion of time series contrast sets, a data mining task that identifies key differences across data groups. We introduced an algorithm to efficiently find time series contrast sets and demonstrated its utility of a host of domains. Many future directions suggest themselves; most obvious among them are extensions to multidimensional time series, to streaming data, and to other distance measures. We will also investigate the possibility of combining the processes of *TS-Diff(T, S, n)* and *TS-Diff(S, T, n)* to avoid redundant computations.

References

[1] Agrawal, R., Imielinski, T. & Swami, A. (1993). Mining Associations Between Sets of Items in Massive Databases. In proceedings of the ACM SIGMOD International Conference on Management of Data. pp. 207-216.

[2] Bay, S. (2000). Multivariate Discretization of Continuous Variables for Set Mining. In proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. Boston, MA. Aug 20-23.

[3] Bay, S. D. & Pazzani, M. J. Detecting Change in Categorical Data: Mining Contrast Sets (1999). In proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. August 15-18. San Diego, CA. 302-306.

[4] Bayardo, J. & Roberto, J. (1998). Efficiently Mining Long Patterns from Databases. In proceedings of 1998 ACM SIGMOD International Conference on Management of Data. pp. 85-93.

[5] Bentley. J. L. & Sedgewick. R. (1997). Fast algorithms for sorting and searching strings. In Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 360-369

[6] He, Z., Xu, X. & Deng, S. (2004). Mining Cluster-Defining Actionable Rules. In proceedings of NDBC '04.

[7] Keogh, E. & Kasetty, S. (2002). On the Need for Time Series Data Mining Benchmarks: A Survey and Empirical Demonstration. In Proc. of SIGKDD. pp 102-111.

[8] Lin, J., Keogh, E., Lonardi, S., & Chiu, B. (2003). A Symbolic Representation of Time Series, with Implications for Streaming Algorithms. In proceedings of the 8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery.

[9] Menzies, T. & Hu, Y (2003). Data Mining for Very Busy People. IEEE Computer, October, pp. 18-25.

[10] Minaei-Bidgoli, B., Tan, P-N., & Punch, W. F. (2004). Mining Interesting Contrast Rules for a Web-based Educational System. In proceedings of 2004 International Conference on Machine Learning Application. Louisville, KY. Dec 16-18.

[11] Sergio Flesca, Giuseppe Manco, Elio Masciari, Luigi Pontieri, Andrea Pugliese (2005). Fast Detection of XML Structural Similarity. IEEE Trans. Knowl. Data Eng. 17(2): 160-175.

[12] Tanaka, Y. & Uehara, K. (2004). Motif Discovery Algorithm from Motion Data. In proceedings of the 18th Annual Conference of the Japanese Society for Artificial Intelligence (JSAI).

[13] Webb, G. (2001). Magnum Opus version 1.3. Computer software, Distributed by Rulequest Research.

[14] Webb, G., Butler, S. & Newlands, D. (2003). On Detecting Differences Between Groups. In Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. Washington DC. Aug 24-27.