The Asymmetric Approximate Anytime Join: A New Primitive with Applications to Data Mining

Lexiang Ye Xiaoyue Wang Dragomir Yankov Eamonn Keogh
University of California, Riverside
{lexiangy, xwang, dyankov, eamonn}@cs.ucr.edu

Abstract

It has long been noted that many data mining algorithms can be built on top of join algorithms. This has lead to a wealth of recent work on efficiently supporting such joins with various indexing techniques. However, there are many applications which are characterized by two special conditions, firstly the two datasets to be joined are of radically different sizes, a situation we call an asymmetric join. Secondly, the two datasets are not, and possibly can not be indexed for some reason. In such circumstances the time complexity is proportional to the product of the number of objects in each of the two datasets, an untenable proposition in most cases. In this work we make two contributions to mitigate this situation. We argue that for many applications, an exact solution to the problem is not required, and we show that by framing the problem as an anytime algorithm we can extract most of the benefit of a join in a small fraction of the time taken by the full algorithm. In situations where the exact answer is required, we show that we can quickly index just the smaller dataset on the fly, and greatly speed up the exact computation. We motivate and empirically confirm the utility of our ideas with case studies on problems as diverse as batch classification, anomaly detection and annotation of historical manuscripts.

1 Introduction.

Many researchers have noted that many data mining algorithms can be built on top of an approximate join algorithm. This has lead to a wealth of recent work on efficiently supporting such joins with various indexing techniques [3], [5], [24], [27]. However, we argue that while the classic database use of approximate joins for record linkage (entity resolution, duplicate detection, record matching, reduplication, merge/purge processing database hardening etc.) does require a full join, many data mining/information retrieval uses of joins can achieve the same end result with an approximate join. Here approximate does not refer to the distance measure or rule used to link two objects, but rather to the fact that only a small subset of the Cartesian product of the two datasets needs to be examined. While the result will not be the same as that of an exhaustive join, it can often be good enough for the task at hand. For example, when performing a classic record linkage, if one dataset contains “`John Lennon, 9 October 1940`”, and the other contains “`John W. Lennon, 09-Oct-40`”, it is clear that these correspond to the same person, and an algorithm that failed to link them would be very undesirable. In contrast, for many data mining applications of joins it is not really necessary to find the nearest neighbor, it can suffice to find a *near-enough* neighbor. Examples of useful tasks that utilize the detection of *near-enough* neighbors as a subroutine include clustering [6], classification [23], anomaly detection [22] and as we show in Section 3.3, historical manuscript annotation. Given this, we show that by framing the problem as an *anytime algorithm* we can extract most of the benefit of the full join algorithm in only a small fraction of the time that it requires. Anytime algorithms are algorithms that trade execution time for quality of results [9]. In particular, an anytime algorithm always has a best-so-far answer available, and the quality of the answer improves with execution time. The user may examine this answer at any time, and then choose to terminate the algorithm, temporarily suspend the algorithm, or allow the algorithm to run to completion. Furthermore, we show that although we are explicitly assuming the data is not indexed at query time, we can build an index on the fly for the smaller dataset and greatly speed up the process.

The rest of the paper is organized as follows. The next section offers more background and explicitly states our assumptions.

1.1 Background and Assumptions.

The *Similarity Join* (SJ) combines two sets of complex objects such that the result contains all pairs of similar objects [3]. It is essentially the classic database join which has been relaxed to allow linkage of two objects that satisfy a similarity criterion. The related *All Nearest Neighbor* (ANN) operation takes as input two sets of multi-dimensional data points and computes for each point in the first set the nearest neighbor in the second set [5]. Note that this definition allows for points in the second set to be unmatched. In this work we introduce the *Asymmetric Approximate Anytime Join* (AAAJ) which also allows objects in the second set to be unmatched, however, it differs from the above in several important ways:

- We assume that the second set is many orders of magnitude larger than the first set. In some cases the second set may be considered effectively infinite[1], for example, this may be the set of all images on the internet or some streaming data.

- The sheer size of the second set means that it cannot be indexed, or can only be "weakly" indexed. For example we cannot index the billions of high dimensional images on the WWW, but we can use Google image search to weakly order images on *size*, date of creation or most significantly (cf. Section 3.3) *keywords* surrounding them.
- The vast majority of work in this area assumes the distance metric used is the Euclidean distance [3], [5], [24], [27]. However, motivated by several real world problems we need to be able to support more general measures such as *Dynamic Time Warping* (DTW), rotation invariant Euclidean distance, or weighted combinations of individual measures such as shape, color and texture similarity.
- Given that the second set may be effectively infinite, we may need to abandon any notion of finding an exact answer; rather we hope to find a high quality answer. In such circumstances we frame the problem as an anytime algorithm.

Note that it is critical to the motivation of this work that we assume that the second set is *not* indexed, because there are many excellent techniques for computing all manner of variations of joins when the data *is* indexed [3], [5], [24], [27]. In addition to the reasons noted above, additional reasons why the data might not be indexed include the following:

- The input query could be intermediate results of complex database queries (as noted in [27]), or the incremental results of a directed web crawl.
- The high dimensionality of the data we need to consider. For example, the five datasets considered in [5] have an average dimensionality of 4.8, the datasets considered in [27] are all two dimensional and even [24] which is optimized for "*handling high-dimensional data efficiently*" considers at most 64 dimensions. In contrast we need to consider datasets with dimensionality in the thousands, and at least some of these datasets are not amiable to dimensionality reduction.

At least some anecdotal evidence suggests that many real world datasets are often not indexed. For example Protopapas et al. [15] have billions of star light curves (time series measuring the intensity of a star) which they mine for

[1]We call the set of images on the WWW "*effectively infinite*" because it is expanded at a rate faster that the download rate of any one machine.

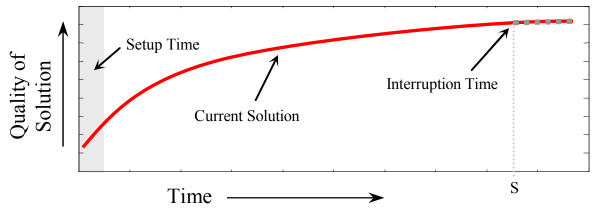


Figure 1: An abstract illustration of an anytime algorithm. Note that the quality of the solution keeps improving up to time S, when the algorithm is interrupted by the user.

outliers, however the data is not indexed due to its high dimensionality and the relative cost and difficulty of building an index for a dataset that may only be queried a few times a year. Additional examples include NASA Ames, which has archived flight telemetry for one million domestic commercial flights. Dr. Srivastava, the leader of the Intelligent Systems Division notes that linear scans on this dataset take two weeks, but the size at dimensionality of the data makes indexing untenable even with state of the art techniques [19]. Given the above, we feel that our assumption that the larger of the datasets is not indexed is a reasonable assumption reflected by many real word scenarios.

The main contribution of this work is to show that joins can be cast as anytime algorithms. As illustrated in Figure 1 anytime algorithms are algorithms that trade execution time for quality of results [9]. In particular, after some small amount of *setup-time* an anytime algorithm always has a best-so-far answer available, and the quality of the answer improves with execution time.

Zilberstein and Russell [28] give a number of desirable properties of anytime algorithms:

- **Interruptability**: After some small amount of setup time, the algorithm can be stopped at anytime and provide an answer.
- **Monotonicity**: the quality of the result is a non-decreasing function of the computation time.
- **Measurable quality**: the quality of an approximate result can be determined.
- **Diminishing returns**: the improvement in solution quality is largest at the early stages of computation, and diminishes over time.
- **Preemptability**: the algorithm can be suspended and resumed with minimal overhead.

As we shall see, we can frame an approximate asymmetric join as any anytime algorithm to achieve all these goals. Due to their applicability to real world problems, there has been increasing interest in anytime algorithms. For example

Algorithm 2.1 BRUTEFORCEJOIN(A, B)

```
1: for $i \leftarrow 1$ to $|A|$ do
2:     $mapping[i].dist \leftarrow \textsc{Dist}(a_i, b_1)$
3:     $mapping[i].pointer \leftarrow 1$
4:     for $j \leftarrow 2$ to $|B|$ do
5:         $d \leftarrow \textsc{Dist}(a_i, b_j)$
6:         if $d < mapping[i].dist$ then
7:             $mapping[i].dist \leftarrow d$
8:             $mapping[i].pointer \leftarrow j$
9: return $mapping$
```

some recent works such as [21] and [23] show how to frame nearest neighbor classification as an anytime algorithm and that top-k queries can also be calculated in an anytime framework, and [25] shows how Bayesian network structure can be learned in an anytime setting.

2 The Asymmetric Approximate Anytime Join.

For concreteness of the exposition we start by formalizing the notion of the All Nearest Neighbor query.

DEFINITION 2.1. Given two sets of objects A and B, an All Nearest Neighbor query, denoted as $ANN\_query(A, B)$, finds for each object $a_i \in A$ an object $b_j \in B$ such that for all other objects $b_k \in B$, $dist(b_j, a_i) \leq dist(b_k, a_i)$.

Note that in general $ANN\_query(A, B) \neq ANN\_query(B, A)$. We will record the mapping from A to B with a data structure called *mapping*. We can discover the index of the object in B that a_i maps to by accessing $mapping[i].pointer$, and we can discover the distance from a_i to this object with $mapping[i].dist$.

It is useful for evaluating anytime or approximate joins to consider a global measure of how close all the objects in A are to their (current) nearest neighbor. We call this Q, the quality of the join and we measure it as: $Q = \sum_{i=1}^{|A|} mapping[i].dist$.

Given this notation we can show the brute force nested loop algorithm for the All Nearest Neighbor (ANN) algorithm in Algorithm 2.1

Note that lines 2 to 3 are not part of the classic ANN algorithm. They simply map everything in A to the first item in B. However, once this step has been completed, we can continuously measure Q as the algorithm progresses, a fact that will be useful when we consider anytime versions of the algorithm below.

In Algorithm 2.1 we have A in the outer loop and B in the inner loop, a situation we denote as BF_AoB (Brute Force, A over B). We could, however, reverse this situation to have B in the outer loop and A in the inner loop. For a batch algorithm this makes no difference to either the efficiency or outcome of the algorithm. Yet, as we shall see, it can make a big difference when we cast the algorithm in an anytime framework.

Before considering our approach in the next section, we will introduce one more idealized strawman that we can compare to. Both flavors of the algorithms discussed above take a single item from one of the two datasets to be joined and scan it completely against the other dataset before considering the next item. Recall, however, that the desirable property of *diminishing returns* would like us to attempt to minimize Q as quickly as possible. For example, assume that we must scan B in sequential order, but we can choose which objects in A to scan across B, and furthermore we can start and stop with different objects from A at any point. Suppose that at a particular point in the algorithm's execution we could either scan a_1 across five items in B to reduce its error from 10.0 to 2.5, or we could scan a_2 across ten items in B to reduce its error from 11.0 to 1.0. The former would give us a *rate* of error reduction of $1.5 = (10.0 - 2.5)/5$, while the latter choice would give us a *rate* of error reduction of $1 = (11.0 - 1.0)/10$. In this case, the former choice gives us the faster rate of error reduction and we should choose it. Imagine that we do this for *every* object in A, at *every* step of the algorithm. This would give us the fastest possible rate of error reduction for a join. Of course, we cannot actually compute this on the fly, we have no way of knowing the best choices without actually doing all the calculations. However, we can compute the best choices offline and imagine that such an algorithm exists. Fittingly, we call such an algorithm *magic*, and can use it as an upper bound for the improvement we can make with our algorithms.

2.1 Augmented Orchard's Algorithm.

While the statement of the problem at hand explicitly prohibits us from indexing the larger dataset B, nothing prevents us from indexing the smaller set A. If A is indexed, then we can simply sequentially pull objects b_j from B and quickly locate those objects in A that are nearer to b_j than to their current best-so-far.

While there is a plethora of choices for indexing dataset A, there are several special considerations which constrain and guide our choice. The overarching requirement is *generality*, we want to apply AAAJ to very diverse datasets, some of which may be very high dimensional, and some of which, for example strings under the edit distance, may not be amiable to spatial access methods. With these considerations in mind we decided to use a slightly modified version of Orchard's algorithm [14], which requires only that the distance measure used be a metric. Orchard's algorithm is not commonly used because its quadratic space complexity is simply untenable for many applications. However, for most of the practical applications we consider here this is not an issue. For example, assume that we can record both the distance between two objects and each of the values in the real valued vectors with the same number of bits. Further we assume

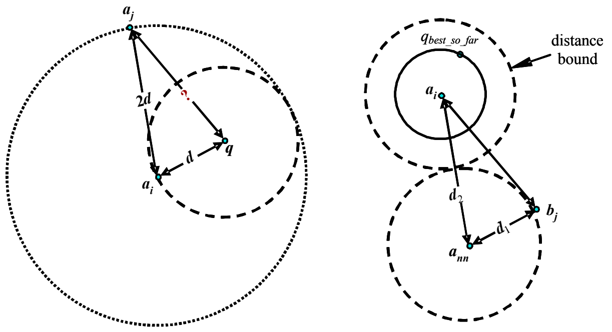


Figure 2: (*Left*) The triangular inequality is used in Orchard's Algorithm to prune the items in A that cannot possibly to be the nearest neighbor of query q. (*Right*) Similarly the triangular inequality is used in Augmented Orchard's Algorithm to prune the items in A that are certain to have a better best-so-far match than the current query q.

have a feature vector length of n per object. Given this, the dataset A itself requires $O(|A|n)$ space, and Orchard index requires $O(|A|^2)$ space. Concretely, for the Butterfly example in Section 3.3 the space overhead amounts to approximately 0.04%, and for the light curve example the overhead is approximately 2.8%. Because Orchard's algorithm is not widely known we will briefly review it in the next section. We refer the interested reader to [14] for a more detailed treatment.

2.1.1 A Review of Orchard's Algorithm. The basic idea of Orchard's algorithm is to quickly prune non-nearest neighbors based on the triangular inequality. In the preprocessing stage the distance between each two items in the dataset A is calculated. As shown in Figure 2 *Left*, given a query q, if the distance between q and an item a_i in dataset A is already known as d, then those items in dataset A whose distance is larger than $2d$ to a_i can be pruned. The distance between these items and q is guaranteed to be larger than d which directly follows the triangular inequality. Therefore, none of them can become the nearest neighbor of q.

Specifically, for every object $a_i \in A$, Orchard's algorithm creates a list $P[a_i].pointer$ which contains pointers to all other objects in A sorted by distance to a_i. I.e., the list stores the index, denoted as $P[a_i].pointer[k]$), of the kth nearest neighbor to a_i within dataset A, and the distance $P[a_i].dist[k]$ between a_i and this neighbor.

This simple array of lists is all that we require for fast nearest neighbor search in A. The algorithm, as outlined in Algorithm 2.2, begins by choosing some random element in A as the tentative nearest neighbor $a_{nn.loc}$, and calculating the distance $nn.dist$ between the query q and that object (lines 1 and 2). Thereafter, the algorithm inspects the objects in list

Algorithm 2.2 ORCHARDS(A,q)

```
1:  $nn.loc \leftarrow$ random integer between 1 and $|A|$
2:  $nn.dist \leftarrow \mathrm{DIST}(a_{nn.loc}, q)$
3:  $index \leftarrow 1$
4:  while $P[a_{nn.loc}].dist[index] < 2 \times nn.dist$ and $index < |A|$ do
5:      $node \leftarrow P[a_{nn.loc}].pointer[index]$
6:      $d \leftarrow \mathrm{DIST}(a_{node}, q)$
7:      if $d < nn.dist$ then
8:          $nn.dist \leftarrow d$
9:          $nn.loc \leftarrow node$
10:         $index \leftarrow 1$
11:     else
12:         $index \leftarrow index + 1$
13: return $nn$
```

$P[a_{nn.loc}].pointer$ in ascending order until one of three things happen. The end of the list is reached, or the next object on the list has value that is more than twice the current *nn.dist* (line 4). In either circumstance the algorithm terminates. The third possibility is that the item in the list is closer to the query than the current tentative nearest neighbor (line 7). In that case the algorithm simply jumps to the head of the list associated with this new nearest neighbor to the query and continues from there (lines 8 to 10).

We can see that a great advantage of this algorithm is its simplicity; it can be implemented in a few dozen lines of code. Another important advantage for our purposes is its generality. The function DIST can be any distance metric function. In contrast, most algorithms use to index data in order to speed up joins explicitly exploit properties that may not exist in all datasets. For example they may require the data to be real valued [5],[27] or may be unable to handle mixed data types.

Note that this algorithm finds the one nearest neighbor to a query q. However, recall that we have a data structure called *mapping* which records the nearest item to each object in A that we have seen thus far. So we need to adapt the classic Orchard's algorithm to update not only the nearest neighbor to q in A (if appropriate) but all objects a_i such that $\mathrm{DIST}(a_i,q) < mapping[i].dist$. We consider a method to adapt the algorithm in the next section.

2.1.2 Augmented Orchard's Algorithm. The results from the previous section together with the mapping structure built for dataset A can be utilized into an extended scheme for approximate joins, which exhibits the properties of an anytime join algorithm too. We term this scheme *Augmented Orchard's Algorithm* (see Algorithm 2.3).

The algorithm starts with an initialization step (lines 1 to 4) during which the table of sorted neighborhood lists $P[a_i]$ is computed. At this point all elements in A are also mapped to the first element in the larger dataset B. To provide for the interruptability property of anytime algorithms, we adopt a B-over-A mapping between the elements of the two sets. The approximate join proceeds as follows: Dataset B is sequentially scanned from the second element on, improving the best-so-far match for some of the elements in dataset A. For this purpose, we first invoke the classic Orchard's algorithm which finds the nearest neighbor $a_{nn.loc}$ to the current query b_j and also computes the distance between them (i.e. $nn.dist$, line 6). If the query improves on the best-so-far match of $a_{nn.loc}$, then we update the element's nearest neighbor as well as its distance (lines 8 and 9).

In addition to improving the best match for $a_{nn.loc}$, the query example b_j may also turn out to be the best neighbor observed up to this point for some of the other elements in A. However, we do not need to check the distances between all a_i and the query b_j. Instead, we can use the pre-computed distance list $P[a_{nn.loc}]$ and once again apply the triangle inequality to prune many of the distance computations. Concretely, we need to compute the actual distance between b_j and any $a_i \in A$ only if the following inequality holds: $mapping[i].dist > |nn.dist - \textsc{Dist}(a_{nn.loc}, a_i)|$. Figure 2 *Right* gives the intuition behind this pruning criterion. The "distance bound" represents the right term of the above inequality. If it is larger or equal to the bound obtained with the best-so-far query to a_i, then the new query b_j cannot improve the current best match of a_i. Note that all terms in the inequality are already computed as demonstrated on line 12 of the algorithm, so no distance computations are performed for elements that fail the triangle inequality. Finally, it is important to point out that a naïve implementation of both Orchard's algorithm and our extension may attempt to compute the distance between the query b_j and the same element a_i more than once, so we keep a temporary structure that stores all elements computed thus far and the corresponding distances for each query b_j. The memory overhead for bookkeeping the structure is negligible and at the same time allows us to avoid multiple redundant computations.

We conclude this section by sketching the intuition behind the pruning performance of the *Augmented Orchard's Algorithm*. Choosing a vantage (center) point is a common technique in many algorithms that utilize the triangular inequality, such as [4], [18], [26], etc. In all of these works one or more center points are selected and the distance between them and all other elements in the dataset are computed. These distances are subsequently used together with the triangular inequality to efficiently find the nearest neighbors for incoming queries. As to what points constitute good centers, i.e. centers with good pruning abilities, depends on several factors [18], e.g. dataset distribution, position of the centers among the other points, as well as the position of the query in the given data space. The two common strategies in selecting the centers are random selection [4], [26] or selection based on some heuristic, e.g. maximal remoteness from any possible cluster center [18]. In the Augmented Orchard's Algorithm we follow a different approach. Namely, for every query b_j we select a different center point $a_{nn.loc}$. This is possible as in the initialization step we have computed all pairwise distances among the elements in A. The algorithm also heavily relies on the fact that the $P[a_i]$ lists are sorted in ascending order of pairwise distance. The assumption is that b_j may impact the best-so-far match only for elements that are close to it, i.e. its nearest neighbor $a_{nn.loc}$ and the closest elements to that neighbor which are the first elements in the $P[a_{nn.loc}]$ list. All remaining elements in this list are eventually pruned by the triangular inequality on line 13 of the algorithm.

Algorithm 2.3 AUGMENTORCHARDS(A,B)

```
1:  for $i \leftarrow 1$ to $|A|$ do
2:      Build $P[a_i]$
3:      $mapping[i].dist \leftarrow \textsc{Dist}(a_i, b_1)$
4:      $mapping[i].pointer \leftarrow 1$
5:  for $j \leftarrow 2$ to $|B|$ do
6:      $nn \leftarrow Orchards(A, b_j)$
7:      if $nn.dist < mapping[nn.loc].dist$ then
8:          $mapping[nn.loc].dist \leftarrow nn.dist$
9:          $mapping[nn.loc].pointer \leftarrow j$
10:     for $index \leftarrow 1$ to $|A|-1$ do
11:         $node \leftarrow P[a_{nn.loc}].pointer[index]$
12:         $bound \leftarrow |nn.dist - P[a_{nn.loc}].dist[index]|$
13:         if $mapping[node].dist > bound$ then
14:             $d \leftarrow \textsc{Dist}(a_{node}, b_j)$
15:             if $d < mapping[node].dist$ then
16:                 $mapping[node].dist \leftarrow d$
17:                 $mapping[node].pointer \leftarrow j$
18: return $mapping$
```

3 Experiments and Case Studies.

In this section we consider examples of applications of AAAJ for several diverse domains and applications. Note that in every case the experiments are completely reproducible, the datasets may be found at http://www.cs.ucr.edu/∼lexiangy/AAAJ/Dataset.html.

3.1 A Sanity Check on Random Walk Data. We begin with a simple sanity check on synthetic data to normalize our expectations. We constructed a dataset of one thousand random walk time series to act as database A, and one million random walk times to act as database B. All time series are of length 128.

Figure 3 shows the rate at which the measure Q decreases as a function of the number of Euclidean distance

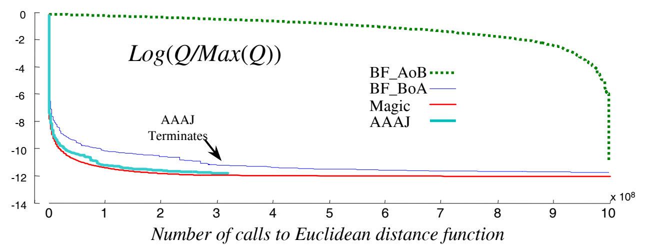


Figure 3: The log of Q as a function of the number of calls to the Euclidean distance.

comparisons made. Note that this figure *does* include the setup time for AAAJ to build the index, but this time is so small relative to the overall time that it cannot be detected in the figure (It can just be seen in Figure 5, where $|B|$ is much smaller relative to $|A|$).

We can see that not only does AAAJ terminate 3 times faster than the other algorithms, but the rate at which it minimizes the error is much greater, especially in the beginning. This of course is the desirable *diminishing returns* property for anytime algorithms. Note in particular that the rate of error reduction for AAAJ is very close to *magic*, which is the optimal algorithm possible, given the assumptions stated for it.

3.2 Anytime Classification of Batched Data. There has been recent interest in framing the classification problem as an anytime algorithm [21], [25]. The intuition is that in many cases we must classify data without knowing in advance how much time we have available. The AAAJ algorithm allows us to consider an interesting variation of the problem which to our knowledge has not been addressed before. Suppose that instead of been given one instance to classify under unknown computational resources we are given a collection of unlabeled instances. In this circumstance we can trivially use AAAJ as an anytime classifier.

Suppose we are given k objects to classify, and we intend to classify them using the One Nearest Neighbor (1NN) algorithm with training set B. However, it is possible that we may be interrupted at any time (Paper [21] discusses several real world application domains where this may happen). In such a scenario it would make little sense to use the classic brute force algorithm in Algorithm 2.1 to classify the data. This algorithm might perfectly classify the first 20% of the k objects before interruption, but then its overall accuracy, including the default rate on the 80% of the data it could not get to in time, would be very poor. Such a situation can arise in several circumstances, for example robot location algorithms are often based on classifying multiple "clues" about location and combining them into a single best guess, and robot navigation algorithms are typically cast as anytime

Table 1: An abridged trace of the current classification of the 30 characters of the stri ng "`SGT PEPPERS LONELY HEARTS CLUB BAND`" vs. the number of distance calculations made

`TTT TTTTTTT TTTTTT TTTTTT TTTT TTTT`	30
`TTT TT`**`I`**`TTTT TTTTTT TTTTTT TTTT TTTT`	436
`TTT TTI`**`I`**`TTT TTTTTT TTTTTT TTTT TTTT`	437
`::: ::::::: :::::: :::::: :::: ::::`	⋮
`S``BT PSPPERS LONEOY REARTF CLUB BANG`	21166
`SBT PSPPERS LONE`**`G`**`Y REARTF CLUB BANG`	*22350
`SBT PSPPERS LONEGY REARTF CLU`**`N`**` BANG`	23396
`::: ::::::: :::::: :::::: :::: ::::`	⋮
`SGT P`**`E`**`PPERS LONELY HEARTZ CLUB HAND`	182405
`SGT PEPPERS LONELY HEART`**`S`**` CLUB HAND`	305794

algorithms [9].

The *Letter* dataset has 26 class labels corresponding to the letters from `A` to `Z`. The task here is to recognize letters given 16 features extracted from image data. There are a total of 20,000 objects in the dataset. We decided on a 30-letter familiar phrase, and randomly extracted 30 objects from the training set to spell that phrase. We ran AAAJ with the 30 letters corresponding to A, and the 19,970 letters remaining in the training set corresponding to B. Table 1 shows a trace of the classification results as the algorithm progresses.

After the first 30 distance comparisons (corresponding to lines 1 to 4 of Algorithm 2.3), every letter in our phrase points to the first letter of dataset B, which happens to be "`T`". Thereafter, whenever an item in A is joined with a new object in B, its class label is updated. When the algorithm terminates after 305,794 distance calculations, the target phrase is obvious in spite of one misspelling ("`HAND`" instead of "`BAND`"). Note, however, that after the AAAJ algorithm has performed just 7.3% of its calculations, its string "`sbt psppers lonegy reartf club bang`" is close enough to the correct phrase to be autocorrected by Google, or to be recognized by 9 out of 10 (western educated) professors at UCR. Figure 4 gives some intuition as to why we can correctly identify the phrase after performing only a small fraction of the calculations. We can see that AAAJ behaves like an ideal anytime algorithms, deriving the greatest changes in the early part of the algorithms run.

In this experiment the improvement of AAAJ over BF_BoA is clear but not very large. AAAJ terminates with 433,201 calculations (including the setup time for the Orchard's algorithms), but the other algorithms only take 598,801. However, this is because the size of A was a mere 30, as we shall see in the next experiment the improvement becomes more significant for larger datasets.

In Figure 5 we see the results of a similar experiment, this time the data is star light curves (discussed in more detail

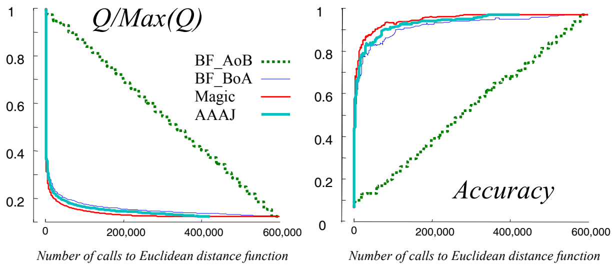


Figure 4: (*Left*) The normalized value of Q as a function of the number of calls to the Euclidean distance. (*Right*) The accuracy of classification, averaged over the 30 unknown letters, as a function of the number of calls to the Euclidean distance.

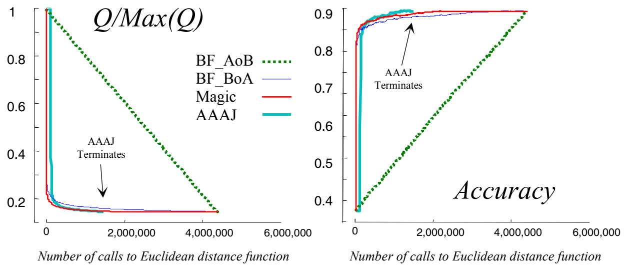
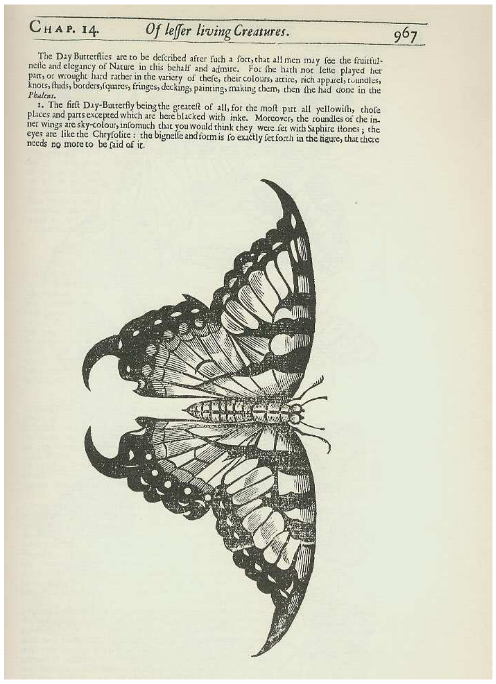


Figure 5: (*Left*) The normalized value of Q as a function of the number of calls to the Euclidean distance. (*Right*) The accuracy of classification, averaged over the 1,000 star light curves, as a function of the number of calls to the Euclidean distance.

in Section 3.4), with $|A|$ = 500 and $|B|$ = 8,736. Each time series was of length 1,024.

At this scale it is difficult to see the advantage of the proposed algorithm. However, consider this. After performing 1,000,000 distance calculations the AAAJ algorithm has reduced the normalized value of Q to 0.1640. On the other hand, the BF_BoA algorithm must perform 4,342,001 distance calculations to reduce the normalized Q to the same level. The following observation is worth noting. While all 4 algorithms have identical classification accuracy when they terminate (by definition), the accuracy of AAAJ is actually slightly higher while it is only 80% completed. This is not an error, it is simply the case that this particular run happened to have a few mislabel objects towards the end of dataset B, and those objects cause several objects in A to be mislabeled.

3.3 Historical Manuscript Annotation. Recent initiatives like the Million Book Project and Google Print Library Project have already archived several million books in digital format, and within a few years a significant fraction of world's books will be online [10]. While the majority of the



CHAP. 14. *Of leſſer living Creatures.* 967

The Day Butterflies are to be deſcribed after ſuch a ſort, that all men may ſee the fruitfulneſſe and elegancy of Nature in this behalf and admire. For ſhe hath not leſſe played her part, or wrought hard rather in the variety of theſe, their colours, attire, rich apparel, roundles, knots, ſtuds, borders, ſquares, fringes, decking, painting, making them, then ſhe had done in the *Phalens*.

1. The firſt Day-Butterfly being the greateſt of all, for the moſt part all yellowiſh, thoſe places and parts excepted which are here blacked with inke. Moreover, the roundles of the inner wings are sky-colour, inſomuch that you would think they were ſet with Saphire ſtones; the eyes are like the Chryſolite: the bigneſſe and form is ſo exactly ſet forth in the figure, that there needs no more to be ſaid of it.

Figure 6: Page 967 of "The Theatre of Insects; or, Lesser living Creatures..." [13], published in 1658, showing the "Day" butterfly.

data will naturally be text, there will also be tens of millions of pages of images. Consider as an example the lithograph of the "Day" butterfly shown in Figure 6.

The image was published in 1658, therefore predating the binomial nomenclature of Linnaeus by almost a century. So a modern reader cannot simply find the butterfly species by looking it up on the web or reference book[2]. However, the shape is well defined, and in spite of being in black and white the author's helpful notes tell us that it is "*for the most part yellowish, those places and parts excepted which are here blacked with inke*".

With a combination of information retrieval techniques and human insight we can attempt to discover the true identify of the species illustrated in the book. Note that a query to Google image search for "Butterfly" returns approximately 4.73 million images (on 9/12/2007). A large fraction of these are not images of butterflies, but of butterfly valves, swimmers doing the butterfly stroke, greeting cards etc. Furthermore, of the images actually depicting butterflies, many depict them in complex scenes that are not amiable to state-of-the-art segmentation algorithms. Nevertheless, a surprising fraction of the images can be automatically segmented with

[2]There is a "Day moth" (*Apina callisto*), however it looks nothing like the insect in question.

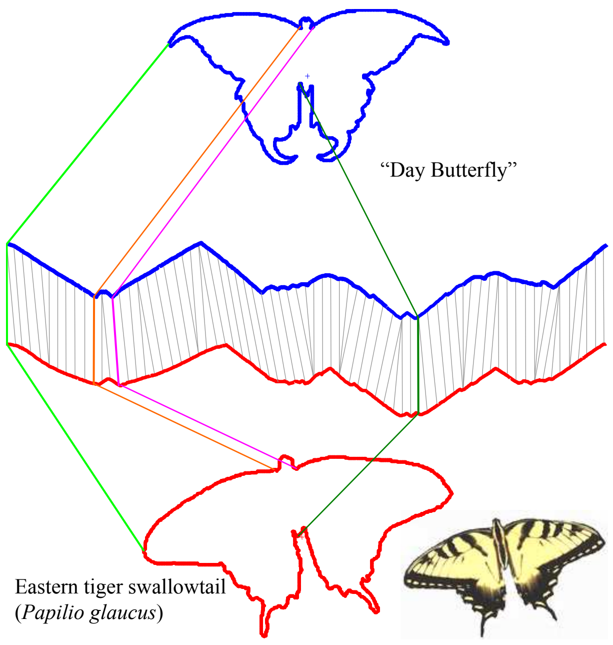


Figure 7: We can compare two shapes by converting them to one-dimensional representations and using an appropriate distance measure, in this case Dynamic Time Warping.

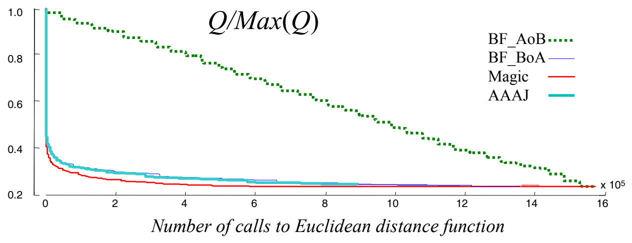


Figure 8: The normalized value of Q as a function of the number of calls to the Euclidean distance.

simple off the shelf tools (we use Matlab's standard image processing tools with default parameters). As illustrated in Figure 7, for those images which can be segmented into a single shape, we can convert the shapes into a one dimensional representation and compare them to our image of interest [11].

While the above considers just one image, there are many online collections of natural history which have hundreds or thousands of such images to annotate. For example, Albertus Seba's 18th century masterwork [17] has more than 500 drawings of butterflies and moths, and there are at least twenty 17th and 18th century works of similar magnitude (see [7], page 86 for a partial list).

We believe that annotating such collections is a perfect application of AAAJ. We have a collection of a few hundred objects, that we want to link to a subset of a huge dataset (the images on the internet). An exhaustive join is not tenable, nor is it needed. We don't need to find the exact nearest match to each butterfly, just one that is near-enough to likely be the same or related species.

Once we have a near-enough neighbor we can use any meta tags or text surrounding the neighbor to annotate our unknown historical images. The basic framework for annotation of historical manuscripts in this way has been the subject of extensive study by several groups, most notably at the University of Padova[3] [1]. However, the work assumes unlimited computation resources and a high degree of human intervention. Here we attempt to show that AAAJ could allow real time exploration of historical manuscripts. We can simply point to an image and right-click and choose annotation-search. At this point the user can provide some text clues to seed the search. In this case we assume the word "butterfly" is provide to the system, which then issues a Google image search.

While there are still some image processing and web crawling issues to be resolved we consider this problem an ideal one to test the utility of AAAJ, so we conducted the following experiment.

Dataset A consists of 35 drawings of butterflies. They are taken from manuscripts as old as 1783 and as recent as 1968. In every case we know the correct species label because it either appears in the text (in some cases in German or French which we had translated by bilingual entomologists) or because the entomologist Dr. Agenor Mafra-Neto was able to unambiguously identify it for us. Data B consist of 35 real photographs of corresponding insects, plus 691 other images of butterflies (including some duplicates of the above) plus 44,215 random shapes. The random shapes come from combining all publicly available shape datasets, and include images of fish, leafs, bones, fruit, chicken parts, arrowheads, tools, algae, and trademark logos. Both datasets are randomly shuffled.

While we used Dynamic Time Warping as the distance measure in Figure 7, it is increasingly understood that for large datasets the amount of warping allowed should be reduced [16], [19]. For large enough datasets the amount of warping allowed should be zero, which is simply the special case of Euclidean distance.

Figure 8 shows the rate of reduction of Q on the butterfly dataset.

While we can see that the rate of error reduction is very fast, it is difficult to get context for the utility of AAAJ from

[3]The work at the University of Padova and related projects consider mostly religious texts, in particular illuminated manuscripts. We know of no other research effort that considers historical scientific texts.

this figure. Consider therefore Figure 9. Here we show in the leftmost column the original historical manuscripts (note that at this resolution they are very difficult to differentiate from photographs). After 10% of the eventual time had passed we took a snapshot of the current nearest neighbors of A. Figure 9 (center) shows the top eight, as measured by $mapping[i].dist$. In the rightmost column we show the final result (which is the same for all algorithms). It is difficult to assign a metric to these results, since precision/recall or accuracy cannot easily be modified to give partial credit for discovering a related species (or a mimic, or convergently evolved species). However, we can intuitively see that much of the utility of conducting this join is captured in the first 10% of the time.

3.4 Anytime Anomaly Detection. In certain domains data observations come gradually over time and are subsequently accumulated in large databases. In some cases the input data rate may be very high or data may come from multiple sources, which makes it hard to guarantee the quality of the stored observations. Still we may need to have some automatic way to efficiently detect whether incoming observations are normal or represent severe outliers, with respect to the data already in the database. A simple means to achieve this is to apply the nearest neighbor rule and to find out whether there is some similar observation stored so far. Depending on the accumulated dataset size and the input rate, processing all incoming observation in online manner with the above simple procedure may still be intractable. At the same time it is often undesirable and, as we demonstrate below, unnecessary to wait for the whole process to finish and then run offline some data cleaning procedure. What we can use instead is an anytime method that computes the approximate matches to small subsets of elements in a batch mode, before the next subset of observation has arrived. Below we demonstrate how our AAAJ algorithm can help us with this.

As a concrete motivating example consider star light curves, also known as variable stars. The American Association of Variable Star Observers has a database of over 10.5 million variable star brightness measurements going back over ninety years. Over 400,000 new variable star brightness measurements are added to the database every year by over 700 observers from all over the world [12], [15]. Many of the objects added to the database have errors. The sources of these errors range from human error, to malfunctioning observational equipment, to faults in punch card readers (for attempts to archive decade old observations) [12]. An obvious way to check for errors is to join the new tentative collection (A) to the existing database (B). If any objects in A are unusually far from their nearest neighbor in B then we can single them out for closer inspection. The only problem with this idea is that a full join on the 10.5 million object database

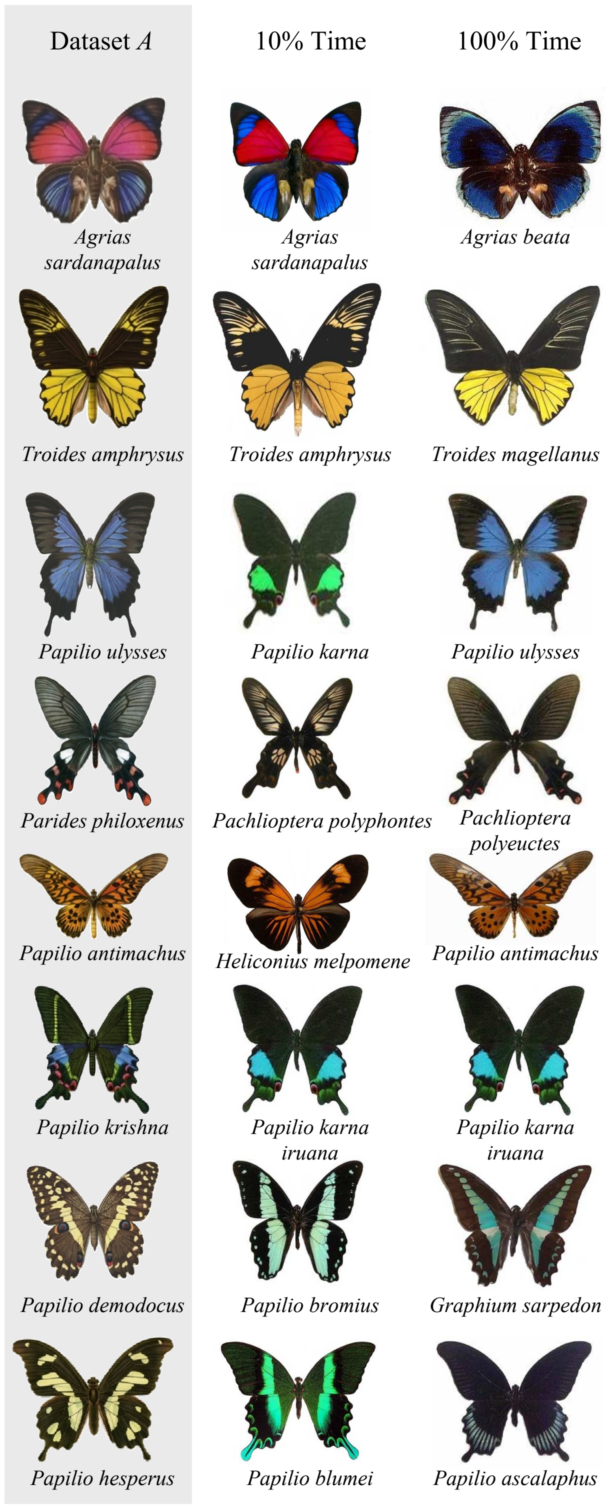


Figure 9: (*Left column*) Eight sample images from the dataset, a collection of butterfly images from historical manuscripts. (*Center column*) The best matching images after AAAJ has seen 10% of the data. (*Right column*) The best matching images after AAAJ has seen all of the data.

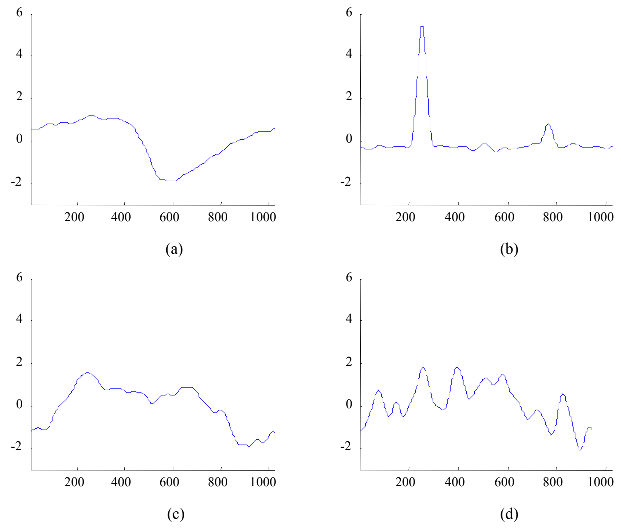


Figure 10: Various light curve vectors in the dataset. (a) and (b) are normal ones. And (c) and (d) are outliers.

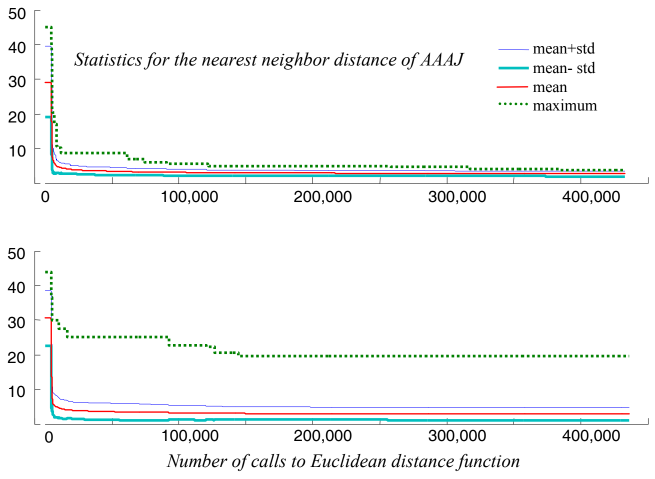


Figure 11: Average nearest neighbor distance with respect to the performed Euclidean distance comparisons. (*Top*) A is composed only of normal elements. (*Bottom*) A contains one outlier.

takes much longer than a day. As we shall see, doing this with AAAJ allows us to spot potentially anomalous records much earlier.

For this experiment we use a collection of star light curves donated by Dr. Pavlos Protopapas of Time Series Center at Harvard's Initiative for Innovative Computing. The dataset that we have obtained contains around 9,000 such light curves from different star classes. Each example is a time series of size 1,024. Figure 10 shows four different light curve shapes in the data. Suppose that a domain expert considers as outliers examples whose nearest neighbor distance is more than a predefined threshold of t standard deviations away from the average nearest neighbor distance [15]. For the light curve data we set $t = 5$ standard deviations, which captures most of the anomalies as annotated by the expert astronomers. Examples (c) and (d) in Figure 10, show two such light curves, while (a) and (b) are less than 5 standard deviations from their nearest neighbors and therefore are treated as normal.

Now assume that the light curve observations are recorded in batch mode 100 at a time. We can apply the AAAJ algorithm to every such set of incoming light curves (representing dataset A in the algorithm) and find its approximate match within the database (set B) interrupting the procedure before the next subset of light curves arrives. If the maximal approximate nearest neighbor distance in A is more than the threshold of 5 standard deviations from the average nearest neighbor distance, then before combining A with the rest of the database we remove from it the outliers that fail the threshold.

To demonstrate the above procedure we conduct two experiments. In the first one dataset A consists of 100 randomly selected examples among the examples annotated as normal. For the second experiment, we replace one of the normal examples in A (selected at random) with a known outlier. This outlier is not believed to be a recording or transcription error, but an unusual astronomical object. Figure 11 shows the improvement in the average nearest neighbor distance in dataset A, again as a function of the performed Euclidean distance computations. The maximal nearest neighbor distance is also presented in the graphs. The top graph corresponds to the experiment where dataset A is composed only of normal elements, and the bottom one is for the dataset with one outlier. The experiment shows how efficiently the AAAJ algorithm can detect the outliers. The average distance drops quickly after performing only a limited number of computations. After that the mean value stabilizes. At this point we can interrupt the algorithm and compute the deviation for each element in A. The elements that fail the expert provided threshold t are likely to fail it had we performed a full join too. In the second example we are able with high confidence to isolate the outlier in only a few thousand distance computations.

4 Conclusions and Future Work

In this work we have argued that for many data mining problems an approximate join may be as useful as an exhaustive join. Given the difficulty of figuring out exactly "how approximate" is sufficient, we show that we can cast joins as anytime algorithms, in order to use as much computational resources as are available. We demonstrated the utility of our ideas with experiments on diverse domains and problems.

As we have shown for the domains considered we can extract most of the benefit of a join after doing only a small fraction of the work (the diminishing returns requirement of [28]), and we can use internal statistics (as in Section 3.4) to achieve the measurable quality requirement [28]. The interruptability and preemptability requirements are trivially true, we can stop the AAAJ algorithm at anytime and we only need to save the relatively tiny mapping data structure if we want to pick up where we stopped at a later date. Finally, with regards to Zilberstein and Russell's requirements, the monotonicity property is clearly observed in Figures 3, 4, 5, 8 and 11.

Future work includes extending Orchard's algorithm to non-metric measures and an enormously scaled up version of the butterfly dataset which also considers color and texture information.

Acknowledgements

We would like to thank all the donors of datasets. In particular, Dr. Agenor Mafra-Neto for his entomological assistance, and Dr. Pavlos Protopapas for his help with the star light curve dataset.

References

[1] M. Agosti, N. Ferro and N. Orio, *Annotations as a Tool for Disclosing Hidden Relationships Between Illuminated Manuscripts*, AI* IA 2007, Proc. of 10th Congress of the Italian Association for Artificial Intelligence. Artificial Intelligence and Human-Oriented Computing , pp. 662–673, 2007.

[2] B. Arai, G. Das, D. Gunopulos, and N. Koudas, *Anytime measures for top-k algorithms*, VLDB 2007, Proc. of 33rd international onference on Very Large Data Bases, pp. 225–237, 2007.

[3] C. Böhm and F. Krebs, *High performance data mining using the nearest neighbor join*, ICDM 2002, Proc. of 2nd International Conference on Data Mining, pp. 43–50, 2002.

[4] W. Burkhard and R. Keller, *Some approaches to best-match file searching*, Communications of the ACM, 16 (1973), pp. 230–236.

[5] Y. Chen and J. M. Patel, *Efficient evaluation of all-nearest-neighbor queries*, ICDE 2007, Proc. of IEEE 23rd International Conference on Data Engineering, pp. 1056–1065, 2007.

[6] C. Elkan, *Using the triangle inequality to accelerate k-means*, ICML 2003, Proc. of International Conference on Machine Learning, 2003.

[7] P. Smart, *The illustrated encyclopedia of the butterfly world*, in Salamander Books London, London, 1975.

[8] K. Fujisawa, S. Hayakawa, and T. Aoki, *Real time search for autonomous mobile robot using the framework of anytime algorithm*, AROB 4th'99, Proc. of 4th Int Symp on Artificial Life and Robotics , pp. 291–296, 1999.

[9] J. Grass and S. Zilberstein, *Anytime algorithm development tools*, ACM SIGART Bulletin, 7 (1996), pp. 20–27

[10] M. Herwig, *Google's total library: putting the world's books on the web*, http://www.spiegel.de/international/, 2007.

[11] E. Keogh, L. Wei, X. Xi, S. H. Lee, and M. Vlachos, *LB_Keogh supports exact indexing of shapes under rotation invariance with arbitrary representations and distance measures*, VLDB 2006, Proc. of 32nd international conference on Very Large Data Bases-Volume 32, pp. 882–893, 2006.

[12] K. H. Malatesta, S. J. Beck, G. Menali, and E. O. Waagen, *The AAVSO Data Validation Project*, The Journal of the American Association of Variable Star Observers, 2006.

[13] T. Mouffet, *The theater of insects.*, Volume 2, Da Capo Press, New York, USA, 1958.

[14] M. T. Orchard, *A fast nearest-neighbor search algorithm*, ICASSP'91, Proc. of International Conference on Acoustics, Speech, and Signal Processing, pp. 2297–2300, 1991.

[15] P. Protopapas, J. M. Giammarco, L. Faccioli, M. F. Struble,and R. Dave, and C. Alcock, *Finding outlier light curves in catalogues of periodic variable stars*, Monthly Notices of the Royal Astronomical Society, 369 (2006), pp. 677–696.

[16] C. A. Ratanamahatana, and E. Keogh, *Three myths about dynamic time warping data mining*, SDM'05, Proc. of SIAM International Conference on Data Mining, 2005.

[17] A. Seba, *Locupletissimi rerum naturalium thesauri accurata descriptio Naaukeurige beschryving van het schatryke kabinet der voornaamste seldzaamheden der natuur*, Amsterdam, 4 vols. 394 B pp. 26–29, vol.3, 1734-1765.

[18] M. Shapiro, *The choice of reference points in best-match file searching*, Communications of the ACM, 20 (1977), pp. 339–343.

[19] Y. Shi, and T. Mitchell, and Z. Bar-Joseph, *Inferring pairwise regulatory relationships from multiple time series datasets*, Bioinformatic, 23 (2007), Oxford Univ Press, pp. 755–763, 2007.

[20] A. Srivastava, Personal Communication, 2007.

[21] K. Ueno, X. Xi, E. Keogh, and D. J. Lee, *Anytime Classification Using the Nearest Neighbor Algorithm with Applications to Stream Mining*, ICDM 2006, Proc. of 6th International Conference on Data Mining, pp. 623–632, 2006.

[22] L. Wei, E. Keogh, and X. Xi, *A Saxually explicit images: finding unusual shapes*, ICDM 2006, Proc. of 6th International Conference on Data Mining, pp. 18–22, 2006.

[23] X. Xi, E. Keogh, C. Shelton, L. Wei, and C. A. Ratanamahatana, *Fast time series classification using numerosity reduction*, ICML 2006, Proc. of 23rd international conference on Machine learning, pp. 1033–1040, 2006.

[24] C. Xia, H. Lu, B. C. Ooi, and J. Hu, *GORDER: An Efficient Method for KNN Join Processing*, VLDB04, Proc. of 30th International Conference on Very Large Data Bases, pp. 756–767, 2004.

[25] Y. Yang, G. Webb, K. Korb, and K. M. Ting, *Classifying under computational resource constraints: anytime classification using probabilistic estimators*, Machine Learning, 69 (2007),pp. 35–53.

[26] P. N. Yianilos, *Data structures and algorithms for nearest neighbor search in general metric spaces*, SODA'93, Proc. of the 4th annual ACM-SIAM Symposium on Discrete algorithms, pp. 311–321, 1993.

[27] J. Zhang, N. Mamoulis, D. Papadias and Y. Tao, *All-nearest-*

neighbors queries in spatial databases, SSDBM 2004, Proceedings. 16th International Conference on Scientific and Statistical Database Management, pp. 297–306, 2004.

[28] S. Zilberstein and S. Russell, *Approximate reasoning using anytime algorithms*, Imprecise and Approximate Computation, Kluwer Academic Publishers, 11 (1995).