

Autocannibalistic and Anyspace Indexing Algorithms with Applications to Sensor Data Mining

Lexiang Ye

Xiaoyue Wang

Eamonn Keogh

Agenor Mafra-Neto¹

Dept. of Computer Science & Eng.
University of California, Riverside, USA

¹ISCA Technologies, Riverside, California 92517

{lexiangy, xwang, eamonn}@cs.ucr.edu, ¹president@iscatech.com

ABSTRACT

Efficient indexing is at the heart of many data mining algorithms. A simple and extremely effective algorithm for indexing under any metric space was introduced in 1991 by Orchard. Orchard's algorithm has not received much attention in the data mining and database community because of a fatal flaw; it requires quadratic space. In this work we show that we can produce a reduced version of Orchard's algorithm that requires much less space, but produces nearly identical speedup. We achieve this by casting the algorithm in an anyspace framework, allowing deployed applications to take as much of an index as their main memory/sensor can afford. As we shall demonstrate, this ability to create an anyspace algorithm also allows us to create *auto-cannibalistic* algorithms. Auto-cannibalistic algorithms are algorithms which initially require a certain amount of space to index or classify data, but if unexpected circumstances require them to store additional information, they can dynamically delete parts of themselves to make room for the new data. We demonstrate the utility of auto-cannibalistic algorithms in a fielded project on insect monitoring with low power sensors, and a simple autonomous robot application.

Keywords

Indexing, Anyspace Algorithms, Data Mining, Sensors.

1. INTRODUCTION

Efficient indexing is at the heart of many data mining algorithms. A simple and extremely effective algorithm for indexing under any metric space was introduced in 1991 by Orchard [11]. The algorithm is commonly known as Orchard's algorithm, however Charles Elkan points out that a nearly identical algorithm was proposed by Hodgson in 1988 [9], [5]. While Orchard's algorithm is currently used in some specialized domains such as vector quantization [6] and compression [20], it has not received much attention in the data mining and database community because of a fatal flaw; it requires quadratic space. In this work we show that we can produce a reduced version of Orchard's algorithm which requires much less space, but

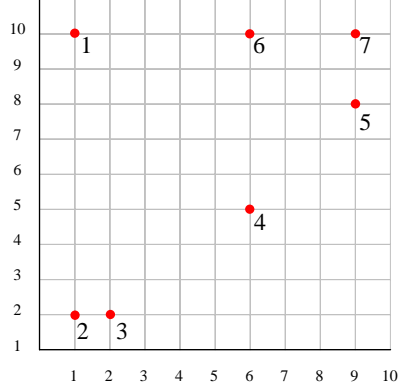
produces nearly identical speedup. We further show that we can cast our ideas in an *anyspace* framework [21], allowing deployed applications to take as much of an index as their main memory/sensor can afford. It is important to note that our reduced space algorithms produce *exactly* the same results as the full algorithm, they simply trade freeing up (a lot of) space for (very little) reduction in speed.

As we shall demonstrate, the ability to cast indexing as an anyspace algorithm allows us to create *auto-cannibalistic* algorithms. Auto-cannibalistic algorithms are algorithms which initially require a certain amount of space to index or classify data, but if unexpected circumstances require them to store additional information, they can dynamically delete ("eat") parts of themselves to make room for the new data. This allows the algorithm to be extremely efficient for a given memory allocation at the beginning of its life, and then gracefully degrade as it encounters outliers which it must store. We demonstrate the utility of auto-cannibalistic algorithms in a fielded project on insect monitoring with low power sensors and show that it can greatly extend the battery life of field deployed sensors.

The rest of the paper is organized as follows. Section 2 reviews the classic Orchard's algorithm and Section 3 introduces our extensions and modifications. We conduct a detailed empirical evaluation in Section 4, and in Section 5 we consider two concrete applications, including one which is already being tested in the field. We conclude with a discussion of related and future work in Section 6.

2. CLASSIC ORCHARD'S ALGORITHM

In order to help the reader understand Orchard's algorithm, and our extensions and modifications to it, we will introduce a simple example dataset in Figure 1 as a running example.



Dataset A

	X	Y
a_1	1	10
a_2	1	2
a_3	2	2
a_4	6	5
a_5	9	8
a_6	6	10
a_7	9	10

Figure 1: A small dataset A containing 7 items, used as a running example in this work

The preprocessing for Orchard’s algorithm requires that we build for each item a_i in our dataset A, a sorted list of its neighbors, annotated with the actual distances to a_i in ascending order. We denote a sorted list for instance a_i as $P[a_i]$, and the full set of these lists for the entire dataset A as $P[A]$. Table 1 shows this data structure for our running example. Note that even for our small running example, the size of the ranked lists data structure is considerable, and would clearly be untenable for real world problems.

Table 1: Orchard’s Algorithm Ranked Lists $P[A]$

Item	1 st NN {dist}	2 nd NN {dist}	3 rd NN {dist}	4 th NN {dist}	5 th NN {dist}	6 th NN {dist}
a_1	6 {5.0}	4 {7.1}	2 {8.0}	7 {8.0}	3 {8.1}	5 {8.2}
a_2	3 {1.0}	4 {5.8}	1 {8.0}	6 {9.4}	5 {10.0}	7 {11.3}
a_3	2 {1.0}	4 {5.0}	1 {8.1}	6 {8.9}	5 {9.2}	7 {10.6}
a_4	5 {4.2}	3 {5.0}	6 {5.0}	2 {5.8}	7 {5.8}	1 {7.1}
a_5	7 {2.0}	6 {3.6}	4 {4.2}	1 {8.2}	3 {9.2}	2 {10.0}
a_6	7 {3.0}	5 {3.6}	1 {5.0}	4 {5.0}	3 {8.9}	2 {9.4}
a_7	5 {2.0}	6 {3.0}	4 {5.8}	1 {8.0}	3 {10.6}	2 {11.3}

The basic intuition behind Orchard’s algorithm is to prune non-nearest neighbors based on the triangular inequality [5]. Suppose we have a dataset $A = \{a_1, a_2, \dots, a_{|A|}\}$, in which we want to find the nearest neighbor of query q . Further suppose a_i is the nearest neighbor found in A thus far. For any unseen element a_j , which will not be the nearest neighbor if:

$$\text{dist}(a_j, q) \geq \text{dist}(a_i, q) \quad (2.1)$$

Given that we are dealing with a metric space, the principle of triangular inequality [5] illustrated in Figure 2 applies here:

$$\text{dist}(a_i, a_j) \leq \text{dist}(a_i, q) + \text{dist}(a_j, q) \quad (2.2)$$

Combining (2.1) and (2.2), we can derive the fact that if:

$$\text{dist}(a_i, a_j) \geq 2 \times \text{dist}(a_i, q) \quad (2.3)$$

is satisfied, a_j can be pruned, since it could not be the nearest neighbor. This is how a combination of the

triangular inequality and the information stored in the ranked lists data structure $P[A]$, can allow us to prune some items a_j without the expense of calculating the actual distance between q and a_j .

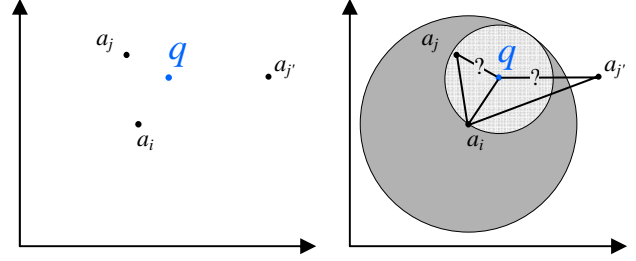


Figure 2: (above, left) Assume we know the pairwise distances between a_i , a_j and a_j . A newly arrived query q must be answered. (above, right) After calculating the distance $\text{dist}(q, a_i)$ we can conclude that items with a distance to a_i less than or equal to $2 \times \text{dist}(q, a_i)$ (i.e. the gray area) might be the nearest neighbor of q , but everything else, including a_j in this example, can be excluded from consideration

The algorithm, as outlined in Table 2, begins by choosing some random element in A as the tentative nearest neighbor. It records the index of the element in $nn.loc$ and calculates the distance $nn.dist$ between q and the element $a_{nn.loc}$ in lines 1 and 2. Thereafter, the algorithm inspects the items in list $P[a_{nn.loc}]$ in ascending order until one of three things happen. If either the end of the list is reached, or the next item on the list has value that is more than twice the current $nn.dist$ (line 4), the algorithm terminates and returns $a_{nn.loc}$ as the nearest neighbor and the corresponding distance $\text{dist}(a_{nn.loc}, q)$. The third possibility is that the item in the list is closer to the query than the current tentative nearest neighbor (line 9). 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 10 to 12).

Table 2: Orchard’s Algorithm Search

```

Function [nn.loc, nn.dist] = Orchard(P[A], q )
1 nn.loc = random_integer_in_range_of(1, |A|)
2 nn.dist = dist(a_{nn.loc}, q)
3 index = 1
4 While P[a_{nn.loc}].dist[index] < 2 * nn.dist AND index < |A| do
5   node = P[a_{nn.loc}].node[index]
6   If node is not yet tested then
7     d = dist(a_{node}, q)
8     If d < nn.dist then
9       nn.dist = d
10      nn.loc = node
11      index = 1
12     Else
13       index = index + 1
14     EndIf
15   Else
16     index = index + 1
17   EndIf
18 EndWhile

```

Note that as the algorithm is traversing down the lists, it may encounter an item more than once. To avoid redundant calculations, a record is kept of all items encountered thus far, and an $O(n)$ hash is sufficient to check if we have already calculated the distance to that item (line 7).

As the reader can appreciate, the algorithm has the advantages of simplicity, and being completely parameter free. Furthermore as shown both here (cf. Section 4) and elsewhere [3], [12], [20], it is a very efficient indexing algorithm. However, while we typically would be willing to spare the quadratic *time* complexity to build the ranked lists dataset, the quadratic *space* complexity has all but killed interest in this method. In the next section we show how we can achieve the same efficiency in a fraction of the space.

3. ANYSPACE ORCHARD'S ALGORITHM

In this section we begin by giving the intuition behind our idea for an anyspace Orchard's algorithm, and then show concrete approaches to allow us to create such an algorithm.

3.1 Truncated Orchard's Algorithm

We motivate our ideas with a simple observation. Note that in Figure 1 the two data items a_2 and a_3 are very close together. As a result, their lists of nearest neighbors in Table 1 are almost identical. This is a redundancy; most queries that are efficiently pruned by a_2 would also be pruned efficiently by a_3 , and vice-versa. We can exploit this redundancy by deleting one entire list, thus saving some space. For the moment let us assume we have randomly chosen to delete the list of a_3 , as shown in Table 3.

Table 3: Truncated Orchard's Algorithm

T	1 st NN	2 nd NN	3 rd NN	4 th NN	5 th NN	6 th NN
a_1	6 {5.0}	4 {7.1}	2 {8.0}	7 {8.0}	3 {8.1}	5 {8.2}
a_2	3 {1.0}	4 {5.8}	1 {8.0}	6 {9.4}	5 {10.0}	7 {11.3}
a_3	goto a_2					
a_4	5 {4.2}	3 {5.0}	6 {5.0}	2 {5.8}	7 {5.8}	1 {7.1}
a_5	7 {2.0}	6 {3.6}	4 {4.2}	1 {8.2}	3 {9.2}	2 {10.0}
a_6	7 {3.0}	5 {3.6}	1 {5.0}	4 {5.0}	3 {8.9}	2 {9.4}
a_7	5 {2.0}	6 {3.0}	4 {5.8}	1 {8.0}	3 {10.6}	2 {11.3}

Note that we cannot just delete the entire row. We have a small amount of bookkeeping to do. It is possible that as we are using the index and traversing down the one of the other lists, we will encounter a_3 and find that it is the *best-so-far*. We should therefore jump to the list for a_3 and continue searching, however the list was deleted. To solve this problem we need to place a special "goto" pointer which tells the search algorithm that it should continue searching from a_2 's list instead.

As the reader will have already guessed, we can iteratively use this idea to delete additional lists, thus saving more space. In the limit, we will be left with a single list as shown in Table 4.

Table 4: Highly Truncated Orchard's Algorithm

Item	1 st NN	2 nd NN	3 rd NN	4 th NN	5 th NN	6 th NN
a_1	goto a_6					
a_2	goto a_4					
a_3	goto a_2					
a_4	5 {4.2}	3 {5.0}	6 {5.0}	2 {5.8}	7 {5.8}	1 {7.1}
a_5	goto a_4					
a_6	goto a_7					
a_7	goto a_5					

Note that whatever algorithm we use to delete lists, we must make sure that we don't end up with cycles. For example if a_2 's row says "goto a_3 " and if a_3 's row says "goto a_2 " we have an infinite loop. Another important observation is that although we should not expect this highly truncated Orchard's algorithm to perform as well as the quadratic space version, we still have not (necessarily) degraded to sequential search. Queries that happen to land near a_4 will be answered quickly, no matter which random starting position we choose.

3.2 Anyspace Orchard's Algorithm

As framed in the previous section, we appear to have a solution to the quadratic complexity of Orchard's algorithm. We can simply work out how much memory is available for our application, and delete the necessary number of lists to make our Truncated Orchard's Algorithm fit. However, there may be situations where the amount of memory is variable (we discuss such applications in more detail in Section 5). In these applications we may find it useful to delete additional lists on-the-fly, as memory becomes more precious. For example, an autonomous robot could use a 90% Truncated Orchard's Algorithm to efficiently classify the items it sees as *friend*, *foe* or *unknown*. For both the *friend* and *foe* categories, it suffices to count how many it encountered. However, for the *unknown* category we may want the robot to store a picture of the unidentified item for later analysis. In this case, it would be useful to throw out additional lists of the Truncated Orchard's Algorithm in order to make space for the new image. An obvious question is, which lists should we toss out? A random selection would be easy, but this may decrease indexing efficiency greatly. Can we do better than random?

Our solution is to frame the Truncated Orchard's Algorithm as an *anyspace* algorithm [21]. Anyspace algorithms are algorithms that trade space for quality of results. In general, an anyspace algorithm is able to solve the problem at hand with any amount of memory, and the speed at which it can solve the problem improves if more space is made

available. In our particular case, we assume we *start* with all the memory of full data structure for the Truncated Orchard's Algorithm, and if we need space to store information about an unexpected event, we “cannibalize” a part of the Truncated Orchard's Algorithm's space to store it. We call such an approach an *Auto-Cannibalistic* algorithm. Figure 3 shows an idealized anyspace algorithm.

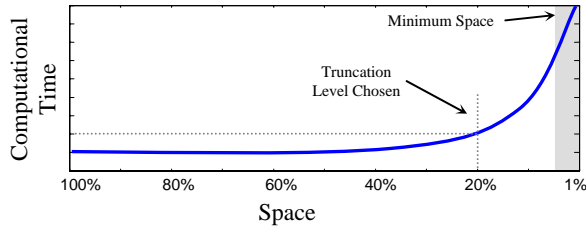


Figure 3: An idealized Anyspace Indexing Algorithm

Note that in this hypothetical case we get the best indexing performance if we use all the memory, however we can throw away 80% of the data and the performance only gets twice as bad. We could instead have thrown away 50% of the data with no significant difference in performance. Note that all anyspace algorithms have some absolute minimum amount of memory which they require. In our case this is the $O(|A|)$ space for the list of items in the dataset.

Assume the size of the full data structure is denoted as unity. Then we can denote the size of an anyspace algorithm as S , where $minimum_space \leq S \leq 1$. In our particular problem we have $O(|A|) \leq S \leq O(|A|^2)$.

The basic framework for using an anyspace algorithm is as follows. We precompute the full space truncated Orchard's Algorithm table and store it in main memory. At some point in the future we expect to get requests for the index which are space limited. For example, sensor **A** may need the index, but only have 2MB available. We simply pull off the best 2MB version and give it to sensor **A**. If sensor **B** requests a 3 MB version, we pull off the best 3MB version and give it to **B**.

Note that for any memory size S of the anyspace algorithm, the data structure S is a proper subset of the data structure $S+e$. That is to say that a larger data structure is always the same as a smaller one, plus some additional data. This is a useful property. First of all it ensures that the size of the full data structure ($S = 1$) is no greater than the original (non anyspace) version (plus a tiny overhead). Thus we have no space overhead for keeping the data in an anyspace format. Second, it allows progressive transmission of the data structure. For example, in our scenario above, if sensor **A** manages to free up some addition memory, and can now devote 2.5MB to indexing the data, we only need to send it the 0.5MB difference.

Anyspace algorithms are rare in machine learning/data mining applications [2], [4], however *anytime* algorithms, which are exact analogues that substitute time instead of space as the critical quality, have seen several data mining applications [13], [14], [15], [17]. Zilberstein and Russell give a number of desirable properties of anytime algorithms, which we can adapt for anyspace algorithms. Below we consider the desirable properties of anyspace algorithms, placing the desirable properties for anytime algorithms in parentheses:

- **Interruptability:** After some small amount of minimum space (*setup time*), the algorithm returns an answer using any additional amount of space (*time*) given.
- **Monotonicity:** the quality of the result is a non-decreasing function of space (*computation time*) used (cf. Figure 3).
- **Measurable quality:** the quality of an approximate result can be determined. In our case, this quality is the indexing efficiency, which can be measured by the number of distance calculations required to answer a query.
- **Diminishing returns:** the improvement in solution efficiency (*quality*) is largest at the early stages of computation, and diminishes as more space (*time*) is given (cf. Figure 3).
- **Preemptability:** The algorithm can use the space given, or additional space if it become available (*the algorithm can be suspended and resumed*) with minimal overhead.

As we shall see, our anyspace indexing algorithm meets all these properties.

Table 5 shows our proposed Anyspace Orchard's algorithm data structure. It differs from the classic data structure (as shown in Table 1) in just two ways.

Table 5: Anyspace Orchard's Ranked Lists (I)

goto list	Item	1 st NN {dist}	2 nd NN {dist}	3 rd NN {dist}	4 th NN {dist}	5 th NN {dist}	6 th NN {dist}
Linear	a_4	5 {4.2}	3 {5.0}	6 {5.0}	2 {5.8}	7 {5.8}	1 {7.1}
goto a_4	a_7	5 {2.0}	6 {3.0}	4 {5.8}	1 {8.0}	3 {10.6}	2 {11.3}
goto a_4	a_3	2 {1.0}	4 {5.0}	1 {8.1}	6 {8.9}	5 {9.2}	7 {10.6}
goto a_4	a_1	6 {5.0}	4 {7.1}	2 {8.0}	7 {8.0}	3 {8.1}	5 {8.2}
goto a_7	a_6	7 {3.0}	5 {3.6}	1 {5.0}	4 {5.0}	3 {8.9}	2 {9.4}
goto a_3	a_2	3 {1.0}	4 {5.8}	1 {8.0}	6 {9.4}	5 {10.0}	7 {11.3}
goto a_7	a_5	7 {2.0}	6 {3.6}	4 {4.2}	1 {8.2}	3 {9.2}	2 {10.0}

First, the rows are no longer in the original order, but sorted in a best first order. Second, note that in the leftmost column there is a small amount of additional information in the form of a *goto list*. This list tells us what to do if we need to free up some space by deleting lists. We will

always delete the lists from the bottom, and replace the entire list by the corresponding *goto* pointer.

As a concrete example, suppose that we must free up approximately 40% of the space used. As shown in Table 6, we can achieve this by deleting the last three lists and replacing them with their respective *goto* pointers.

Table 6: Anyspace Orchard's Ranked Lists (II)

goto list	Item	1 st NN {dist}	2 nd NN {dist}	3 rd NN {dist}	4 th NN {dist}	5 th NN {dist}	6 th NN {dist}
<i>Linear</i>	a_4	5 {4.2}	3 {5.0}	6 {5.0}	2 {5.8}	7 {5.8}	1 {7.1}
<i>goto</i> a_4	a_7	5 {2.0}	6 {3.0}	4 {5.8}	1 {8.0}	3 {10.6}	2 {11.3}
<i>goto</i> a_4	a_3	2 {1.0}	4 {5.0}	1 {8.1}	6 {8.9}	5 {9.2}	7 {10.6}
<i>goto</i> a_4	a_1	6 {5.0}	4 {7.1}	2 {8.0}	7 {8.0}	3 {8.1}	5 {8.2}
	a_6	<i>goto</i> a_7					
	a_2	<i>goto</i> a_3					
	a_5	<i>goto</i> a_7					

3.3 Constructing Anyspace Orchard's

Assume we have truncated Orchard's data structure, T . At one extreme, T has all lists $P[a_i]$ for $1 \leq i \leq |A|$, and is thus the "classic" Orchard's data structure. At the other extreme, it has only a single list, and we can only efficiently answer queries that happen to be near the untruncated item.

Assume that we have a black box function $evaluate_addition(T, i)$ which given T returns the estimated utility of adding list $P[a_i]$. This function estimates the expected number of items that an arbitrary query must be compared to in order to find its nearest neighbor by adding list $P[a_i]$ to the existing table, and returns the highest utility with the smallest comparison number. For the moment we will gloss over the details of this function, except to note that it allows us to create an Anyspace Orchard's Algorithm. The basic idea of the algorithm is to start with an empty set T , and iteratively use function $evaluate_addition(T, i)$ to decide which list to add next. For example, we can see from column 2 of Table 5, the lists were added in this order: $a_4, a_7, a_3, a_1, a_6, a_2$ and a_5 .

The formal algorithm shown in Table 7 is divided into two phases; selection and mapping. The first phase is a simple, greedy-forward selection search. T is initialized as an empty stack in line 1. For each outer iteration, the algorithm tests every item (that was not previously selected) with the utility function (line 7), picks the item with highest utility and pushes it onto T (lines 8 to 10). This procedure continues until all the items are pushed onto T . In essence, this phase sorts all the items by their expected utility for indexing. For instance, in the running example shown in Table 5, it first selects a_4 , then a_7 , then a_3 , etc.

The second phase is to create the *goto* list. Our strategy is to start from the bottom of the table, repeatedly selecting

the candidate with the lowest utility, and change its *goto* pointer to point at its nearest neighbor with a higher utility.

Table 7: Build Anyspace Orchard's Construction

```

Function [goto_list, P'] = BuildAnyspaceOrchards(A, P)
1  T = NULL
2  For i = 1 to |A|
3    max_eval = evaluate_addition(T, i)
4    additem = 1
5    For j = 2 to |A|
6      If  $a_j$  is candidate
7        eval = evaluate_addition(T, j)
8        If (eval > max_eval)
9          max_eval = eval
10         additem = j
11      EndIf
12    EndIf
13  EndFor
14  push(T, additem)
15 EndFor
16 P' = sort P best first according to T
17 For i = |A| to 2
18   For j = 1 to |A| - 1
19     index = P[ai].pointer[j]
20     If  $a_{index}$  appears above  $a_i$  in P'
21       goto_list[ai] =  $a_{index}$ 
22       break;
23     EndIf
24   EndFor
25 EndFor

```

To achieve this we scan the sorted Orchard's algorithm table bottom up (as in line 17). For each item a_i under consideration, we scan down its nearest neighbor list $P[a_i].pointer$ in lines 18 and 19. If we find one nearest neighbor a_{index} that ranks above a_i in sorted Orchard's algorithm table, we assign a_{index} to the entry of a_i in the *goto* list $goto[a_i]$ in line 20 to 22. In the running example, we first consider the item a_5 . Following a_5 's nearest neighbor list, we check the item a_7 , which is on the second line of the sorted Orchard's algorithm table, and thus above a_5 . We therefore make $goto[a_5]$ point to a_7 . We next consider the item a_2 , and so on.

We have yet to explain how our $evaluate_addition$ function is defined. We propose a simple approach that takes both *density* and *overlap* of each item into consideration. Intuitively, we assume the density distribution of the queries to be at least somewhat similar to the density distribution of the data in the index, so we want to reward items for being in a dense part of the space. At the same time, if we have one item from the center of a dense region, then there is little utility in having another item from the same region (overlap), so we want to penalize for this.

Concretely, our algorithm works as follows: the candidate's pool is initialized to include all the items. Given the parameter nearest neighbors number n , we set the item a_i maximum utility in the candidate pool with smallest distance between a_i and its n^{th} valid nearest neighbor. We then delete a_i from candidate pool. In addition, for all the

items that on the a_i 's nearest neighbor list, ranked from 1 to n , we assign them the minimum utility value (overlap penalty) and they are never again considered to be neighbors of any other items. Suppose we have m items initially, in our approach, we first pick $\lfloor m/(n+1) \rfloor$ *pivot* items according its radius to cover n valid nearest neighbors. After that, for those $m\%(n+1)$ items that are uncovered by any pivot item, we pick them in random order. Finally, we pick remaining nearest neighbors items in random order. We did consider several other possibilities, such as leaving-one-out evaluation, measuring the rank correlation, mutual information, or entropy gain between two lists as a measure of redundancy. However either these ideas did not work empirically, or required several parameters to be tuned. As a simple sanity check, we will include empirical comparisons to random, a variant of *evaluate_addition* which simply chooses a random list to add.

3.4 Using Anyspace Orchard's Algorithm

After constructing the sorted Orchard's algorithm table, it is easy to adapt the Orchard's algorithm search technique (Table 2) to allow it to search in the truncated version. The main task is to decide what we should do if the algorithm indicates a jump to the list of a certain item a_i while that list has already been deleted.

Simply jumping to the list of a_j if $goto[a_i] = a_j$ is not possible, as the list of a_j may also have been deleted. Consider the running example in Table 6. If two more lists are deleted, the Orchard's Algorithm table shown in Table 8 is produced.

Suppose some query arrives, and the algorithm finds itself needing to jump to the list of a_2 . Since the list of a_2 is deleted, it wants to jump to a_3 which the *goto* entry of a_2 indicates. However, it cannot do so, because the list of a_3 is also deleted. The algorithm should continue the jump action, and see whether the list of $a_4 = goto[a_3]$ is deleted. The general approach to find a valid item to jump to is described in Figure 9.

Table 8: Anyspace Orchard's Ranked Lists (III)

goto list	Item	1 st NN {dist}	2 nd NN {dist}	3 rd NN {dist}	4 th NN {dist}	5 th NN {dist}	6 th NN {dist}
<i>Linear</i>	a_4	5 {4.2}	3 {5.0}	6 {5.0}	2 {5.8}	7 {5.8}	1 {7.1}
<i>goto a_4</i>	a_7	5 {2.0}	6 {3.0}	4 {5.8}	1 {8.0}	3 {10.6}	2 {11.3}
	a_3	<i>goto a_4</i>					
	a_1	<i>goto a_4</i>					
	a_6	<i>goto a_7</i>					
	a_2	<i>goto a_3</i>					
	a_5	<i>goto a_7</i>					

Table 9: Find Valid goto for Single Item

```

Function valid_goto = Find_goto(goto_list, a_i)
1  item = a_i
2  While 1
3    item = goto_list[item]
4    If the list of item is not deleted
5      break
6    EndIf
7  EndWhile
8  Return item

```

There are two additional things we need to check. One is if the list of a_j has not been visited, and the other is if the distance between q and a_j is smaller than the distance between query q and a_{nn} , which is the item whose list we are currently visiting.

The first test is performed to avoid an infinite loop which makes the algorithm jump back and forth between the ranked lists. The second item is because the spirit of Orchard's algorithm tells us to attempt to jump to the item that is nearer the query than the item being visited. After confirmation of the two questions above, we can safely jump to the list of a_j . Table 10 shows the entire Anyspace Orchard's search algorithm.

Table 10: Anyspace Orchard's Algorithm Search

```

Function nn = AnyspaceOrchards(A, q)
1  Build P
2  [goto, P'] = BuildAnyspaceOrchards(A, P)
3  truncate P' from bottom to fit it into memory
4  nn.loc = random_interger_in_range_of(1, |A|)
5  nn.dist = dist(a_{nn.loc}, q)
6  index = 1
7  bestpos = nn.loc
8  bestdist = nn.dist
9  While P[a_{nn.loc}].dist[index] < 2 * nn.dist AND index < |A|
10  item = P[a_{nn.loc}].pointer[index]
11  d = dist(a_{item}, q)
12  If d < nn.dist
13    If d < bestdist
14      bestdist = d
15      bestpos = item
16    EndIf
17  a_goto = valid_goto(goto_list, a_{index})
18  If list of a_goto not visited AND dist(a_goto, q) < nn.dist
19    nn.loc = goto
20    nn.dist = dist(a_goto, q)
21    If nn.dist < bestdist
22      bestdist = nn.dist
23      bestpos = nn.loc
24    EndIf
25  Else
26    index = index + 1
27  EndIf
28  Else
29    index = index + 1
30  EndIf
31  EndWhile

```

First, the algorithm checks the available space and builds the truncated sorted Orchard's algorithm table in lines 1 to 3. One modification is that we add some bookkeeping to the item that best matches query, and the corresponding distance in lines 7 and 8, lines 13 to 15, and lines 21 to 23. The reason is the list of best-match item may have been deleted, thus the search does not necessarily end at the best-match item. Therefore the information of the best-

match item should be stored at the time we compare the item to the query. Another modification is that we find the valid item to jump to in line 16, and test if we should jump to that item in line 17 as discussed above. Apart from these minor changes, the rest of the algorithm is exactly the same as in the classical Orchard's algorithm.

3.5 An Optimization of Anyspace Orchard's

As described above, the mechanism used to create the Anyspace Orchard's algorithm may be quite slow. In some sense this is not a big issue, since we expect to perform this step offline. Nevertheless, it is reasonable to ask if we can speed up this process.

Note that one cause of its lethargy is the redundant calculations spent in finding the valid *goto* entries. The time will increase as the more lists being deleted. Here we show a simple one-scan strategy to update the entire *goto* list and avoid this overhead, which is described in Table 11. The parameter *cut* means the number of sorted rank lists can accommodate in the memory.

Table 11: Find Entire Valid *goto* List

	Function valid_goto_list = find_goto_list(P', goto_list, cut)
1	For i = 1 to cut
2	a _i = the item on the i th row of P'
3	valid_goto_list[a _i] = a _i
4	EndFor
5	For i = cut+ 1 to n
6	a _i = the item on the i th row of P'
7	valid_goto_list[a _i] = valid_goto_list[goto_list[a _i]]
8	EndFor

We initialize the *goto* entry of those items which have *not* been truncated to point to themselves in lines 1 and 4. This initialization does not affect these items, as they never use *goto* pointers, and makes finding valid *goto* pointers easier for the remaining items. In lines 5 and 8, we consider these truncated items from top to bottom. For each item *a_i* we are considering, we are sure all the items whose position above it in the table already point to a valid item. Suppose *a_k* is the item that *a_i*'s original *goto* pointer points to. In that case, the option is either making the valid *a_i*'s *goto* pointer the same as *a_k*'s *goto* pointer when *a_k* is truncated, or when *a_k* is *not* truncated, the pointer should point to *a_k*. Once the valid *goto* list is built, we can avoid all the redundant *goto* searches.

Another optimization is to narrow down the pruning criteria. We discovered an extra inequality we can exploit using the distance between the query and the best-so-far item. As in Figure 4.A, suppose *a_j* is the best-so-far item while its rank list has been truncated, and *a_i* is a valid item which *a_j*'s *goto* pointer points to. As shown in Figure 4.B, we only need to compute the actual distance between *q* and any *a_k* ∈ *A* only if the following inequality holds: $dist(a_i, a_k) < 2 \times dist(a_j, query) + dist(a_i, a_j)$. Recall that, as shown in Section 2, *a_k* can be pruned if

$$dist(a_j, a_k) \geq 2 \times dist(a_j, query) \quad (3.1)$$

However, $dist(a_j, a_k)$ is not available because $P[a_j]$ was truncated. There is an additional inequality between the three items where we can have the lower bound of the value of $dist(a_j, a_k)$:

$$dist(a_j, a_k) \geq dist(a_i, a_k) - dist(a_i, a_j) \quad (3.2)$$

Combining (3.1) and (3.2), if we have

$$dist(a_i, a_k) - dist(a_i, a_j) \geq 2 \times dist(a_j, query) \quad (3.3)$$

item *a_k* can be admissibly pruned. In our implementation, we can simply replace line 8 of Table 10 with the line below, which is

8new	While $P[a_{nn.loc}].dist[index] < 2nn.dist$ AND $index < A $ AND $dist(a_i, a_k) < 2dist(a_j, query) + dist(a_i, a_j)$
------	--

In general, on most datasets, this optimization improves indexing efficiency by 10% to 30%, so we use it in all experiments that follow.

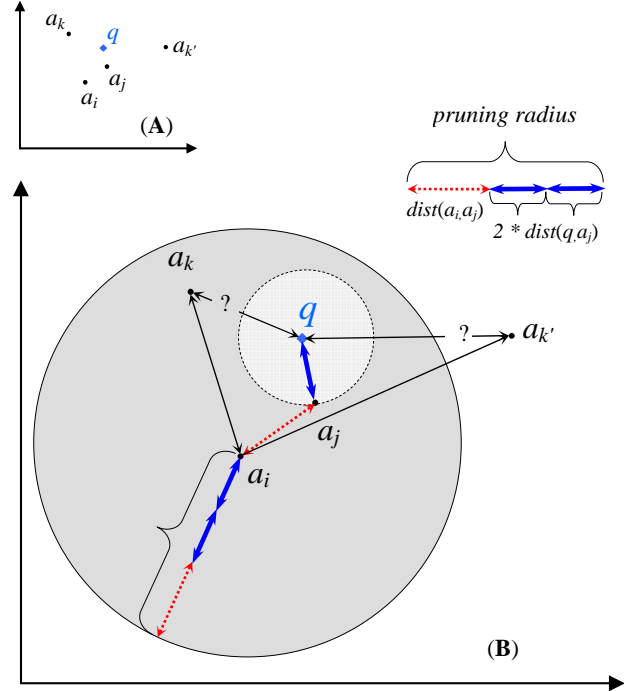


Figure 4: Assume *a_j* is the current nearest neighbor of query *q*, and that $P[a_j]$ was deleted and replaced with the *goto* entry *a_i*, (A) A newly arrived query *q* must be answered. (B) An admissible pruning rule is to exclude items whose distance to *a_i* is greater than or equal to $dist(a_i, a_j) + 2 \times dist(q, a_j)$. In this example, everything outside the large gray circle can be pruned

4. EXPERIMENTS

We begin by stating our experimental philosophy. In order to ensure easy replication of our work we have placed all data and code at a publicly available website [18].

Recall that our algorithm for constructing truncated Orchard's algorithm has a parameter n . One objective of our experiments is to see how sensitive our algorithm is to the choice of this parameter. A further objective is test the utility of our *evaluate_addition* function. It might be that *any* function would work well in this context. As a simple baseline comparison we compare against a function that randomly orders the lists for truncation. To understand the algorithm's efficiency we measure the average number of distance calculations needed to answer a one nearest neighbor query. In this, and all subsequent experiments we normalize the range to be between zero and one when creating figures, so a perfect algorithm would have a value near zero, and a sequential scan would have a value of one.

We begin with a simple experiment on a synthetic dataset. We created a dataset of 5,000 random items from a 2D Gaussian distribution. We created a further 50,000 test examples. We begin by testing the indexing efficiency of the full Orchard's algorithm (i.e with zero truncation) with the test set, and then we truncate a single item and test again. We repeat the process until there is only a single list available to the algorithm, Figure 5 shows the experiment on the synthetic data.

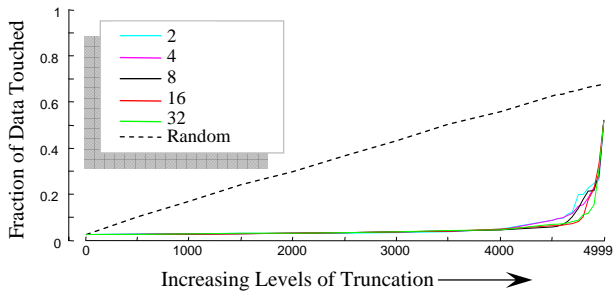


Figure 5: The indexing efficiency vs. level of truncation for a synthetic dataset

The results appear very promising (compare to the idealized case in Figure 3). First, it is clear that the choice of parameter n has very little impact on the results. Note that we can truncate 80% of the data without making a significant difference to the efficiency. Thereafter, the efficiency does degrade, but gracefully. Further notice that in contrast to our algorithm, the random approach has linear relationship between size and efficiency. This tells us our *evaluate_addition* function is finding redundancies in the data to exploit.

We next consider a problem of indexing data from a road sensor. This sensor data was collected for the Glendale on-amp at the 5-North freeway in Los Angeles. The observations were taken over 25 weeks, at 5 minute count aggregates. As the location is close to the Dodgers stadium, it has bursty behavior on days in which a game is played. There are a total of 47497 observations, we randomly

choose 1,000 to build our index, and used the rest for testing. Figure 6 shows the results.

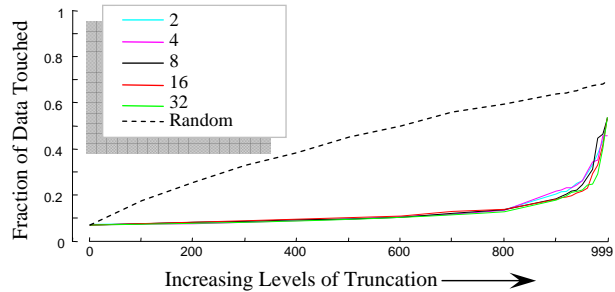


Figure 6: The indexing efficiency vs. level of truncation for the Dodgers dataset

As before, the performance of our algorithm is exactly we would like to see in an idealized anyspace algorithm, and once again, and our algorithms performance is almost invariant to the choice of parameter n .

Having shown that truncated Orchard's algorithm passes some simple sanity checks, in the next section we consider detailed case studies of problems we can solve using our algorithm.

5. EXPERIMENTAL CASE STUDIES

We conclude the experimental section with two detailed case studies of uses for our algorithms.

5.1 Insect Monitoring

ISCA Technologies is a Southern California based company that produces devices to monitor and control insect populations in order to mitigate harm to agricultural and human health. They have produced a “smart-trap” device that can be mass produced, and left unattended in the field for long periods to monitor a particular insect of interest.

The system under consideration here is primarily designed to track *Aedes aegypti* (yellow fever mosquito), a mosquito that can spread the dengue fever, yellow fever viruses, and a host of other diseases. In particular, the system needs to classify the sex of the insects and keep a running total of how many of each sex are encountered¹. In order to only capture *Aedes aegypti*, the trap can be designed specifically for them. For example, carbon dioxide can be used as an attractant (this eliminates most non-mosquito insects). The trap can be placed at a certain height which eliminates low flying insects, and the entrance can be made small enough to prevent larger insects from entering. Nevertheless, as we shall see, non *Aedes aegypti* insects can occasionally enter the traps.

¹ Recall that only female mosquitoes suck blood from humans and other animals.

While we have attempted classification of the insects with Bayesian Classifiers, SVMs and decision trees, our current best results come from using 1-Nearest Neighbor classification with a 4-dimensional feature vector extracted from the audio signal. A further advantage of using 1NN is that it allows us to come up with a simple definition of outlier. We empirically noted that on average both males and females tend to be a distance of m to their nearest neighbors. This number has a relatively small standard deviation. We therefore have defined an outlier as a data sample that is more than $m + 4$ standard deviations from its nearest neighbor. Figure 7 shows a visual intuition of this.

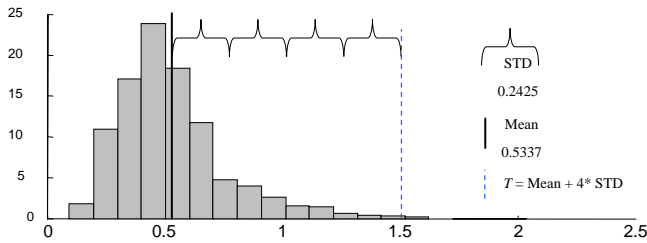


Figure 7: The distribution of distances to nearest neighbor for 1,000 insects in our training set. We consider exemplars whose distance to their nearest neighbor is more than the mean plus 4 standard deviations to be suspicious and worthy of follow-up investigation

There are two obvious sources of outliers; non-insect sounds from outside the trap—including helicopters and farming equipment—and non-*Aedes aegypti* insects that enter the trap. Knowing the true identity of the outliers can be very useful. In the former case, it may be possible to change the traps location to reduce the number of outlier events caused by the sound of farm machinery. In the latter case, it may be useful know which unexpected insects we have caught. For example, we may have been subject to an unexpected invasion, as in the famous invasion of Glassy-winged Sharpshooters (*Homalodisca coagulata*) which almost devastated the wine industry in Temecula southern California’s Temecula Valley 1997 [1]. However, the low power/low memory requirements of the traps prohibit recording the entire audio stream. Our solution therefore is to use an auto-cannibalistic algorithm which allows efficient indexing to support the one-nearest neighbor classification, and to record a one second snippet of outlier sounds. Each snippet requires the auto-cannibalistic to delete 3 lists from its table.

A classifier was built using 1,000 lab reared insects, with 500 of each sex. The training error suggests that we can achieve over 99% accuracy, however we have yet to confirm this by hand annotation of insects captured in the field. In Figure 8 we see a plot showing the indexing efficiency for various levels of truncation.

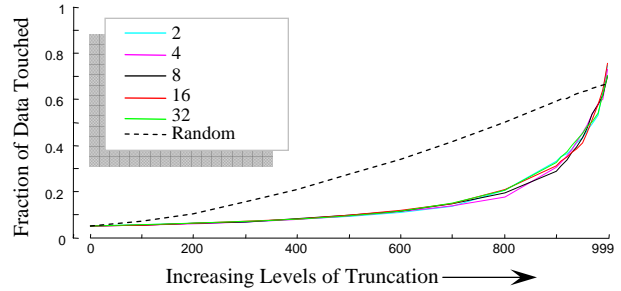


Figure 8: Indexing efficiency vs. space on the insect monitoring problem

Note that the application lends itself well to any space framework. Even when we have deleted 25% of the data we can barely detected any change in the indexing efficiency.

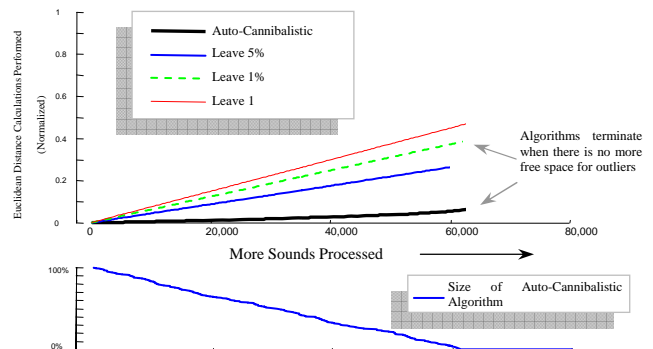


Figure 9: top) The cumulative number of Euclidean distance calculations performed vs. the number of sound events processed. bottom) The size of auto-cannibalistic algorithm vs. the number of sound events processed

Having demonstrated the concept in the lab, we deployed an auto-cannibalistic algorithm in the field. As a baseline comparison we compared to hard-coded truncated Orchard’s algorithms where just 5%, 1% and a single list remains. Figure 9 shows the results.

The figure shows that the more memory an indexing algorithm has, the more efficiently it can process incoming data. The auto-cannibalistic algorithm starts out with a full Orchard’s algorithm in memory and is consequently much more effective than its smaller rivals. Over time, outliers are encountered and must be stored, so the amount of memory available to auto-cannibalistic algorithm decreases, causing it to become less efficient. However it is difficult to detect this for the first 50,000 or so events. As the power required is almost perfectly correlated with the cumulative number of Euclidean distance calculations, which in turn is simply the area under the curves, the auto-cannibalistic algorithm requires less than one tenth the energy of the 5% Orchard’s algorithms, and is able to handle 4.99% more outlier events before its memory fills up.

5.2 Robotic Sensors

In this section we consider the utility of auto-cannibalistic algorithms in a robotic domain. Note that unlike the insect example in the previous section, this is not a mature fielded product, it is simply a demonstration on a toy problem. In particular, we are not claiming that the approach below for finding unexpected tactical sensations is the best possible approach; it is merely an interesting test bed for a demonstration of our ideas.

The Sony AIBO, shown in Figure 10, is a small quadruped robot that comes equipped with a tri-axial accelerometer. This accelerometer measures data at a rate of 125 Hz.

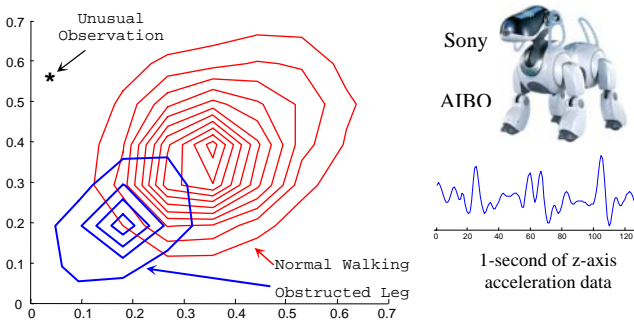


Figure 10: clockwise from top right. A Sony Aibo robot. An on-board sensor can measure acceleration at 125 Hz. The accelerometer data projected into two dimensions

By examining the sensor traces, we can (perhaps imperfectly) learn about the surface the robot is walking on. For example, in Figure 10 we show a two dimensional mapping of the Z-axis time series for both normal unobstructed walking and walking when one leg is obstructed. While the overlap of the two distributions in this figure suggests a high error rate, in three dimensions the separation is better, and we can achieve about 96% accuracy. Naturally it is useful to distinguish between these two situations, as the robot can attempt to free itself or change direction. In addition to merely classifying current state, it may be useful to detect unusual states and photograph them for later analysis. As the AIBO has only 4 megabytes of flash memory on board, memory must be used sparingly. A single compressed image with its 416x320 pixel camera requires about 100k of space.

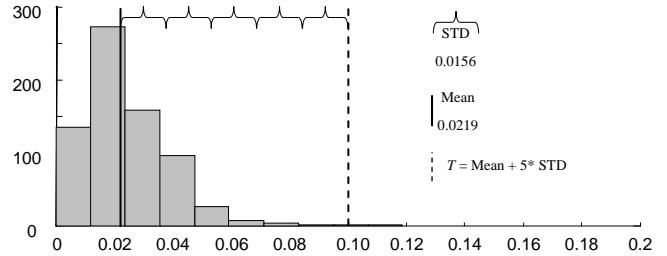


Figure 11: The distribution of distances to nearest neighbor for 700 tactile events in our training set. We consider exemplars whose distance to their nearest neighbor is more that the mean plus 5 standard deviations to be suspicious, and worthy the memory required to take a photograph

We use the same basic framework as in the previous section here. We took 700 training instances and use them to build a 1-nearest neighbor classifier indexed by the truncated Orchard's algorithm. Every time an outlier is detected, and an image must be stored, we must delete an average of 18 lists from our index to make room for it. Figure 12 shows the result of the experiment. As before, we compare to hard-coded truncated Orchard's algorithms where just 5%, 1% and a single list remains.

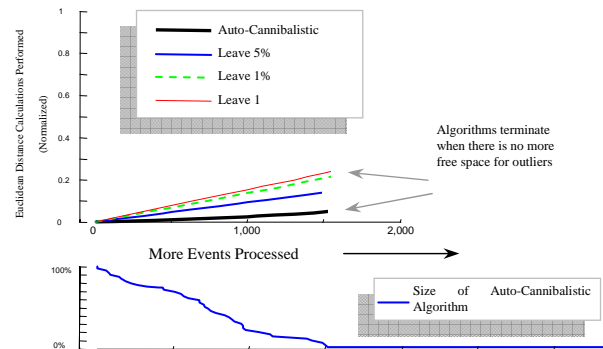


Figure 12: top) The cumulative number of Euclidean distance calculations performed vs. the number of tactile events processed. bottom) The size of auto-cannibalistic algorithm vs. the number of tactile events processed

As with the insect example, the truncated Orchard's algorithm requires only a fraction of the energy of the fixed-size indices, and is able to process data until just a single list remains available after dealing with event 1,522.

6. DISCUSSION

We have introduced a novel indexing method especially for sensor data mining. In this section, we discuss the related work and provide future extensions.

6.1 Related Work

Indexing is important for similarity search because it will reduce a large amount of searching time since it can

eliminate expensive distance calculations. The problem of indexing under metric distance has been studied intensively in the last decade and many efficient algorithms have been proposed. There are basically two alternative categories:

- **Embedding method:** for the objects in the data set of N dimensions, it created a k -dimension feature vector to represent each object. The distance calculated in the k -dimension feature space provides a lower bound of the actual distance between the objects. If k is considerably smaller than N , and the lower bound is reasonably tight, it can prune a lot of objects with much less distance calculation effort. Examples of this method are in [8], [16].
- **Distance-based method:** typical distance-based method is based on partition. All or some distance between the objects in the data set are precomputed. When a query comes in, we can estimate the majority of distances between the objects and query based on a small fraction of the actual distance we have computed between the objects and the query, and thus prune a lot of non-qualified objects. The vantage-point tree method [19] is an example in this category.

The method we proposed in this paper falls into the second category. The obvious drawback of method in the second category is that the index data structure is fixed, which means, the indexing has a rigid memory requirement. However, under the scenario where main memory is bottleneck, e.g. in the sensor or robot, the algorithms with fixed memory requirement may fail.

6.2 Conclusion

In this paper, our major contribution is that:

- We have shown the Orchard's algorithm may be rescued from its relative obscurity by considering it as an anyspace algorithm and leveraging off of its unique properties to produce efficient sensor mining algorithms.
- We have further shown what we believe is the first example of an auto-cannibalistic algorithm.

Future work includes a large scale deployment and testing of the insect sensors, and a more general exploration of the notation of auto-cannibalism for other applications.

Acknowledgements: We thank the staff of ISCA Technologies for their assistance with insect project, and Dr. Manuela Veloso and Douglas Vail for donating the robotic data.

7. REFERENCES

- [1] S.J. Castle, S.E. Naranjo, J.L. Bi, F.J. Byrne and N.C. Toscano. Phenology and demography of *Homalodisca coagulata* (Hemiptera: Cicadellidae) in southern California citrus and implications for management *Bulletin of Entomological Research* (2005) 95, 621–634
- [2] A. Checheta and K. Sycara. An Any-space Algorithm for Distributed Constraint Optimization. In *Proceedings of AAAI Spring Symposium on Distributed Plan and Schedule Management*, March, 2006.
- [3] K.L. Clarkson. Nearest-neighbor searching and metric space dimensions. In *Nearest-Neighbor Methods for Learning and Vision: Theory and Practice*, MIT Press, 2006.
- [4] A. Darwiche. Any-space probabilistic inference. In *16th Conference on Uncertainty in Artificial Intelligence*, pages 133-142, 2000.
- [5] C. Elkan. Using the triangle inequality to accelerate kMeans. In *Proceedings of the Twentieth International Conference on Machine Learning*, pages 147-153, 2003.
- [6] D. Ghosh and A. P. Shivaprasad. Fast codeword search algorithm for real-time codebook generation inadaptive VQ. *Vision, Image and Signal Processing, IEE Proceedings*, volume: 144, issue: 5. pages 278-284, 1997.
- [7] J. Grass and S. Zilberstein. Anytime algorithm development tools. *SIGART Artificial Intelligence*. volumn 7, no. 2, ACM Press, 1996.
- [8] C. Faloutsos and K. I. Lin. FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *Proceedings of the 1st ACM SIGKDD Conference*, pages 163–174, 1995.
- [9] M. E. Hodgson. Reducing computational requirements of the minimum-distance classifier. *Remote Sensing of Environments*, 25:117-128, 1988.
- [10] A. Ihler, J. Hutchins, and P. Smyth. Adaptive event detection with time-varying Poisson processes. In *Proceedings of the 12th ACM SIGKDD Conference (KDD'06)*, 2006.
- [11] M. T. Orchard. A fast nearest-neighbor search algorithm. *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2297-2300, IEEE Computer Society Press, 1991.
- [12] B. C. Song; J. B. Ra. A fast search algorithm for vector quantization using L2-norm pyramid of codewords; *Image Processing, IEEE Transactions*, volume 11, issue 1, pages 10-15, 2002.
- [13] P. Smyth and D. Wolpert. Anytime Exploratory Data Analysis for Massive Data Sets. In *Proceeding of the 3rd International Conference on Knowledge Discovery and Data mining (KDD'97)*, pages 54-60, 1997.

- [14] K. Ueno, X. Xi, E. Keogh and D.J. Lee. Anytime Classification Using the Nearest Neighbor Algorithm with Applications to Stream Mining, *In Proc. of the 6th International Conference on Data Mining (ICDM'06)*, pages 623-632, 2006.
- [15] M. Vlachos, J. Lin, E. Keogh & D. Gunopulos. A Wavelet-Based Anytime Algorithm for K-Means Clustering of Time Series. In Workshop on Clustering High Dimensionality Data and Its Applications. *In the 3rd SIAM Int'l Conference on Data Mining*. San Francisco, CA. 2003.
- [16] J. T. Wang, X. Wang, K. I. Lin, D. Shasha, B. A. Shapiro & K. Zhang. Evaluating a class of distance-mapping algorithms for data mining and clustering. *In Proceedings of the 4th ACM SIGKDD Conference*, pages 307-311, 1999.
- [17] G. I. Webb, Y. Yang, J. Boughton, K. Korb and K. M. Ting. Classifying under computational resource constraints: Anytime classification using probabilistic estimators. *Technical Report 2005/185, Clayton School of Information Technology, Monash University*, 2005.
- [18] L. Ye.(2008). Supporting URL for this paper. www.cs.ucr.edu/~lexiangy/Anyspace/Dataset.html
- [19] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. *In Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 311-321, 1993.
- [20] K. Zatloukal, M. H. Johnson and R. Ladner. Nearest neighbor search for data compression. *In Data Structures, Nearest Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges*. AMS, 2002.
- [21] S. Zilberstein, and S. Russell. Approximate reasoning using anytime algorithms. *In Imprecise and Approximate Computation*, Kluwer Academic Publishers, 1995.