Detecting Data Structures from Traces

Alon Itai Michael Slavkin
Department of Computer Science
Technion Israel Institute of Technology
Haifa, Israel
itai@cs.technion.ac.il mishaslavkin@yahoo.com

July 30, 2007

Abstract

This paper shows how to deduce the data structure that gave rise to a trace of memory accesses. The proposed methodology is general enough to classify different sets of data structures and is based on machine learning techniques.

1 Introduction

1.1 The Problem

The paper discusses the following problem:

> Given a *trace*, i.e., a sequence of address accesses to the data area of a program, detect what is the data structure that gave rise to the trace.

We assume that the set of possible data structures is given in advance. Also, since many data structures have several slightly different implementations (a linked list can be implemented with either a dummy header node or without it; it can also have a dummy end node) each distinct implementation is considered a distinct data structure. In this research we have limited ourselves to the case where the program uses a single static data structure, i.e. it only supports searches (no insertions/deletions) and all searches are to a single data structure.

Our thesis is that each data structure gives rise to a distinct pattern of traces, thus by classifying the traces one can recover the data structure that produced them. Machine learning algorithms were applied to perform the classification: In the *training phase* we extracted features from the traces and then passed them to the Machine Learning algorithm for classification. In the validation phase, we employed the resultant classification algorithm to classify new traces.

We have experimented with a number of popular classification algorithms (C4.5 [6], SVM[11], Naïve Bayes [5]) and compared their accuracy. The decision tree that results from the C4.5 classifier provides us also with an easy to understand algorithm for classifying data structures.

1.2 Motivation

Predicting the behavior of a program's use of memory is of high interest for different level tools, from operating systems and compiler optimization engines to data structure implementations. They all can improve their performance given an effective algorithm for making such predictions.

(a) Current memory hierarchy solutions are based mostly on time and space locality of accesses and the LRU approach for ageing of cache lines (for associative caches) and memory pages. An algorithm for predicting the behavior of memory access at a high level of abstraction might provide much better solutions. For example, the Symmetrix Disk Array (of EMC) [3] used the patterns of accesses to prefetch elements of arrays and matrices.

(b) An operating system or a compiler can optimize memory allocation layout, given the feedback of an algorithm for detection of data structures from a previous run. For example, in case of hashing with chaining we would like to allocate nodes from the same chain at the same physical page. We may also try to activate the data structure detection algorithm on a prefix of the current run to achieve a performance improvement for the rest of the run. The JVM uses Just-in-Time compilation, i.e., the frequently executed parts of the JAVA programs are compiled dynamically, while the less frequently executed JAVA code is interpreted [2]. Thus, by the time of the dynamic compilation the compiler may utilize the feedback of the data structure detection algorithm on a prefix of the current run.

(c) The actual usage of generic data structures can be far from optimal for a specific insertion/deletion/search sequence. For example, choosing the STL class 'list' to implement a buffer can be a bad choice if the stored objects are not processed in LIFO or FIFO order. It is surely poor design to have chosen a list. However, if chosen we would like to have a methodology to detect such cases so that an inappropriate data structure could possibly be replaced online.

2 Problem Definition

"A data structure is a way to store and organize data in order to facilitate access and modifications" [1]. Data is organized as records each consisting of a sequence of fields. A field is either:

(a) a primitive type, or

(b) a pointer to a record, or

(c) one or more sub-records (allowing for recursion).

Following software engineering design principles, we assume that all data structures are encapsulated, i.e., all access to the data structures is via access functions. In particular every search initiates at the root of the tree, head of a list or at an arbitrary place in an array. Consequently, pointers that are not related to the implementation of the data structure are not allowed in this study.

Note that a data structure is characterized not only by its organization but also by its access pattern and the modifications it supports. For example, a heap can be implemented by an array, i.e., its organization is identical to the organization of an array while its operations and hence its access pattern are different.

Since in this paper we restrict our study to searches only (no insertions or deletions) the data structure is static – its structure (topology of records and relations between them) doesn't change throughout the program.

We look at traces of memory accesses (address sequences) resulting from a series of searches to detect the underlying data structure organization and the traversal methods.

If we know the correspondence between an address and the record that resides at that address we can replace each address in the trace by some symbolic representation of the set of corresponding records. The resultant trace is called the *symbolic trace*.

We have thus defined two subproblems: learning the symbolic trace from the address trace, and learning the data structure associated with the symbolic trace. Decoupling the two problems sets a yardstick by which we can measure the effectiveness of the learning method, and the success in deducing the symbolic trace, as well as how are our results affected by the lack of knowledge about the layout structure of the records.

3 The Symbolic Trace

We first examine the symbolic trace and describe an approach to recover the topology of records and the relations between them, as well as the traversal methods.

The association between addresses and records is not one-to-one: an address can correspond to an entire record or its first subrecord. To distinguish between such cases, the symbolic trace is partitioned into levels: the top level being the records themselves, the next level is that of the subrecords etc.

For example, an address may be associated with a matrix entry, a row of the matrix and the matrix as a whole (if it is implemented as one long array). Thus we consider three traces: first level trace for matrices, second level trace rows and third level trace for the matrix elements.

We consider each level of record traces in isolation. Each level corresponds to a specific topology of the subset of records of this level and the subset of relationships between them, as well as the traversal methods. To detect the data structure we combine data from all levels.

Since the data structures are encapsulated, within a level the program can only move from one record to another via a pointer or from a subrecord to another subrecord (an important special case is moving from an array element to the next array element). Since a record and its subrecord belong to different levels, a symbolic trace of a single level does not include such moves.

For any two records A and B we would like to capture the relation "there is a pointer from A to B" from the symbolic trace. However, the trace manifests only the proximity of accesses to two records, which could arise from a sequential access to an array where adjacent elements will be accessed one after the other. Since this access pattern is typical of arrays it may be used to characterize data structures.

To count how many times B appears immediately after A in the trace, we constructed the following binary matrix M:

$$M[A,B] = \begin{cases} 1 & \text{if } A \text{ immediately precedes } B \text{ at least } \theta \text{ times;} \\ 0 & \text{otherwise} \end{cases}$$

θ is a predetermined constant, which in our implementation was equal to 1.

For a record B let its *in-degree* be the number of records A for which $M[A,B] = 1$, and its *out-degree* the number of records C for which $M[B,C] = 1$. Let *in/out-degree* stand for edges in both directions, i.e., for the case where both (A,B) and (B,A) exist. Let us look at the distribution of tiplets of (in-degree, out-degree, in/out-degree) for records of several common data structures:

1. A binary search tree: the distribution consists of the tuples (1,1,0), (1,2,0) for nodes, (1,0,0) for leaves and (0, 2,0) or (0,1,0) for the root.

2. A deterministic skip list: we should have (0, many,0) for the $-\infty$ node, (many,0,0) for the $+\infty$ node and (1,2,0) for most of other nodes.

3. A singly linked list: we have (1,1,0), (0,1,0), (1,0,0).

4. A doubly linked list: the most common tuple is (2,2,2).

Thus the distributions of tuples differ significantly among data structure. Note that even for the same data structure the distribution of the tuples of a single search and multiple searches may differ. For example, for a binary search tree we expect the root to incur (0, 2, 0) or (0, 1, 0). In a trace of a sequence of several searches the in-degree of the root will be much greater than 0, because we return to the root to start the next search.

These degrees have a range from 0 to ∞. Since the exact value of any such degree is significant only when it is small but for large values we need only a qualitative measure, we partitioned the entire range to a finite number of subranges. The records are thus partitioned into equivalence classes depending on the values of the subranges of their (in-degree, out-degree, in/out-degree). We then replaced the occurrence of each record in the trace by its equivalence class.

To capture the time dependency, we looked at short subsequences in the trace. We limited their length to 3, since longer subsequences produced more features than the learning programs could handle, while limiting the length to 1 or 2 produced poor results. The learning of the data structure from the symbolic trace will be discussed in Section 5.

4 Address Trace

To deduce the symbolic trace from the address trace we use clustering algorithms. I.e., we first partitioned the addresses into clusters and then we examined cluster trace – the occurrence of the clusters in the trace.

As in the symbolic trace of Section 3 we would like to partition the cluster trace into *levels*. To achieve this we ran the clustering algorithm with different resolutions (the exact set of resolutions depends on the clustering algorithm and it will be discussed in the next paragraph). The problem of detecting a data structure has thus become a problem of detecting the topology of clusters and relations between clusters, and the traversal methods. As in Section 3 we converted this problem into a set of sub-problems, one for each clustering level. Then we combined data from several sources, one for each trace level, to detect the data structure.

We examined three clustering algorithms:

(a) Single-pass clustering [8] -- a clustering technique that joins addresses A and B into the same cluster if the distance between these addresses is below some threshold. We may calculate such clusters by a single pass through a sorted list of addresses. We partitioned the distances into subranges whose sizes grow as 2^{2^n}.

(b) Agglomerative hierarchical clustering [4] — a clustering technique that starts with singleton clusters (with a single address per cluster) and then merges a number of pairs of clusters one by one. The two clusters with the best similarity measure are selected for merging. The similarity measure that we used is the distance between centroids (mean elements) of the clusters [10]. The number of merges is a function of the clustering resolution.

(c) Paging clustering – an approach that zeroes the least significant bits of an address to calculate the corresponding cluster (page). The number of bits zeroed depends on the resolution, i.e., the cluster (page) size and alignment vary for different clustering resolutions. Note, that the calculation of the cluster of an address is independent of other addresses. This clustering technique is very sensitive to the alignment of records in memory, but it is not aware of the actual distribution of memory accesses. Even though this clustering technique can be very inexact in some cases, it is still interesting to examine its performance because it takes only O(1) time to calculate to which cluster an address belongs.

We proceed with the cluster trace as we did with the symbolic trace.

5 Applying Machine Learning

We examined three popular classification approaches (C4.5, SVM and Naïve Bayes) to classify data structures. SVM and Naïve Bayes provide us only with a classification, while C4.5 results also in a decision tree, which helps us visualize the classification decisions. The common paradigm for the training phase is as follows:

(a) Prepare a set of training examples that contains a number of samples for each data structure.

(b) Convert each example into a set of features.

(c) Train the classifier on a set of pairs of the form (data_structure, set_of_feature_values) and prepare all the files that are necessary for the testing phase (these files constitute the learning model and they are specific to each classifier).

The common paradigm for the testing phase is as follows:

(a) Prepare a set of testing examples that contains a number of samples for each data structure.

(b) Convert each example into a set of features.

(c) Run the classifier on the set of features of each example and output the predicted data structure.

Both the training and the testing examples were generated artificially. To create examples we first generated a sample of the data structure with random parameters and random entry values. Then we generated a series of searches, which are typical for this type of data structure (e.g., searches to random keys that start at the root of the tree, at the head of a singly linked list, at an arbitrary place in an array and so on), applied to the current sample. We recorded the trace of the memory accesses (the address sequence) resulting from this series of searches. In the symbolic trace mode we recorded symbols (which encode information about all the records that correspond to the appropriate address) instead of addresses.

We collected tuples and treated them as words in the Bag-of-Words approach for text categorization [9]. In the Bag-of-Words approach the question of whether taking the existence of a feature or the number of occurrences is called *feature evaluation.* Taking the existence it is called *binary* and the number is called *natural.* Finally, as is customary, we take the logarithm of the number of occurrences instead of the occurrences .

6 The Experiment

6.1 The Setup

We examined classification approaches from the previous chapters for eight data structures (for each data structure we examined searches to several random keys that either belong or do not belong to the data structure):

(a) An AVL tree.

(b) A deterministic skip list.

(c) A doubly linked list with searches, which start from the first or from the last node at random.

(d) A singly linked list.

(e) A hash table with conflict resolution solved by chaining.

(f) A matrix where each search starts at some random displacement from the row (column) start and continues for a random number of steps with a random step length. The direction of the search is also random.

(g) A vector where each search starts at some random displacement from the row (column) start and continues for a random number of steps with a random step length. The direction of the search is also random. We examine two different cases of this data structure:

    i) A vector with a mainly random access pattern, which is produced by a high percentage of long steps (we will quickly go out of the vector boundary with such steps).

    ii) A vector with a mainly sequential access pattern, which is produced by a high percentage of short steps.

We covered all combinations of the next input parameters:

- The type of a classifier.
- The symbolic trace or the address trace with all the clustering algorithms.
- The binary vs. natural feature evaluation.
- The noise threshold.
- Clustering levels (resolutions) supported.

We generated 300 training examples and 300 testing examples per data structure and per each combination of input parameters. The generation of the data structure sample incurs the execution of several memory allocations that are produced by the compiler. It follows an allocation algorithm that might be biased. For example, the Microsoft Visual C++ compiler attempts to allocate all records sequentially in memory. This might lead to a bias: the addresses of consecutive elements in a list might form an arithmetic sequence, and thus will be indistinguishable from an array. For our experiment we tried to avoid such a bias by using of our own memory allocator, which chooses the next allocated address at random.

6.2 An Example

Figure 1 depicts two decision trees which were created by the C4.5 classifier with the binary feature evaluation and the single-pass clustering algorithm.

Fig. 1(a) depicts the decision tree that distinguishes between deterministic skip lists and AVL trees. Multiple searches to an AVL tree cause the root to have high in-degree. In deterministic skip lists both the root and the $+\infty$ node have high in-degree. Word a.2 describes accesses to two distinct nodes with large in-degrees. This can occur only for deterministic skip lists.

Fig. 1(b) depicts the decision tree that distinguishes between doubly linked lists and AVL trees. Nodes (or clusters of resolution 1) with (in-degree, out-degree) = (1, 2) are typical of AVL trees but not of doubly linked lists. If no such a sequence exists (word b.1) we obviously have a doubly linked list.

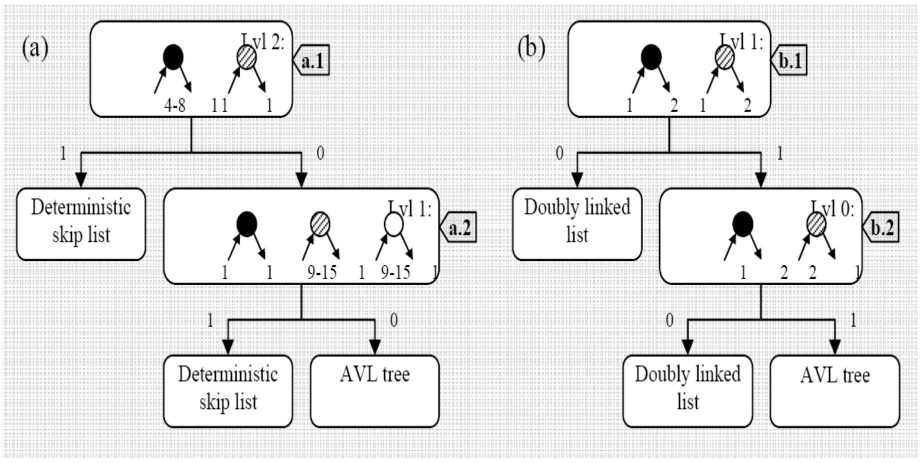
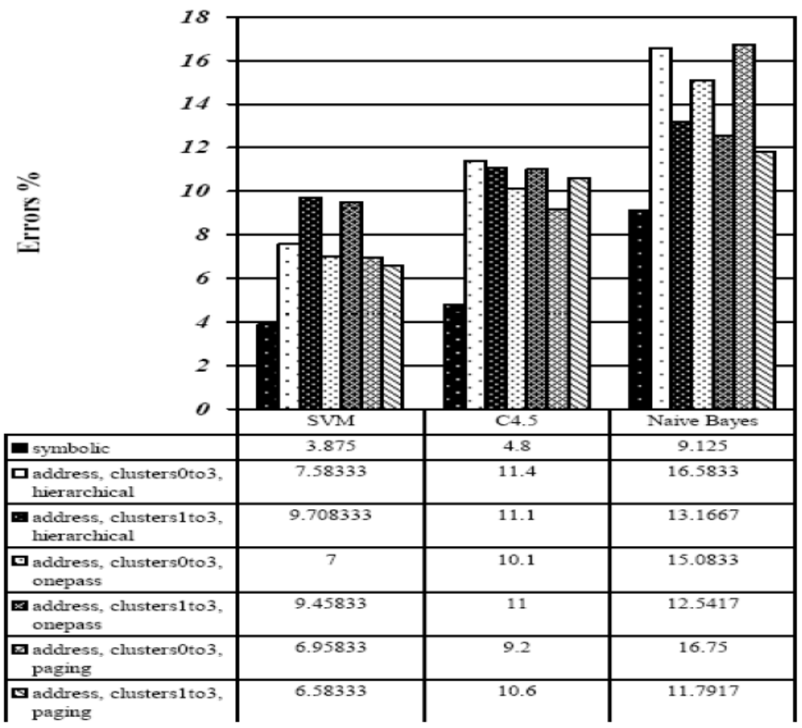


Fig. 1: Decision trees for distinguishing between (a) deterministic skip lists and AVL trees (b) Doubly linked list and AVL tree

6.3 Performance

In Figure 2 we compared the accuracy of classifiers in conjunction with different clustering approaches. These clustering approaches included the three clustering algorithms of Section 4 for the address trace and the symbolic trace approaches. Moreover, for the same clustering algorithm we looked at two different subsets of clustering levels. One subset (clusters 0 to 3) contained the highest resolution (zero resolution) which spreads each address per cluster while the second (cluster 1to 3) did not contain it. The remaining resolutions appeared in both subsets.

Note the relatively poor performance of Naïve Bayes approach. The Naïve Bayes classifier is based on the simplifying assumption that the attribute values are conditionally independent. In our case, there are short words that appear as parts of longer words. Hence their attribute values are dependent.



	SVM	C4.5	Naive Bayes
■ symbolic	3.875	4.8	9.125
□ address, clusters0to3, hierarchical	7.58333	11.4	16.5833
▩ address, clusters1to3, hierarchical	9.708333	11.1	13.1667
▫ address, clusters0to3, onepass	7	10.1	15.0833
▦ address, clusters1to3, onepass	9.45833	11	12.5417
▤ address, clusters0to3, paging	6.95833	9.2	16.75
▧ address, clusters1to3, paging	6.58333	10.6	11.7917

Fig. 2: Natural feature evaluation with different traces (symbolic or address in conjunction with the different clustering algorithms)

As can be seen, the symbolic approach provides us with the best classification accuracy for SVM and C4.5 classifiers. There is not much difference in the performance of the remaining clustering algorithms. It is especially interesting that the accuracy of the paging approach (whose cluster calculation is easy) is in line with other clustering techniques, which take into account the actual distribution of memory accesses.

Note that in most of the cases the clusters 0–3 subset of clustering levels results in a better accuracy than the clusters 1–3 subset. In most of the data structures that we examined the record size is greater than one and there is a specific access pattern for primitive fields of the same record. We miss this access pattern if we do not look at words from the highest resolution level (with a single address per cluster).

The performance of the binary feature evaluation case is very similar to the natural one. We have not included the graphs due to lack of space.

In Table 1 we present the accuracy of the SVM classifier tested on a symbolic trace with the natural feature evaluation. We plot the distribution of actual data structures at a testing phase vs. predicted ones. It should be noted that most

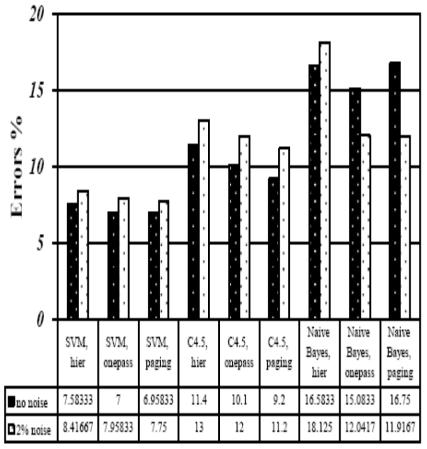
Actual Predicted	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)
(a) AVL tree	297	1	0	0	1	1	0	0
(b) Determinsitc skip list	0	297	0	3	0	0	0	0
(c) Doubly linked list	0	0	291	9	0	0	0	0
(d) Singly linked list	0	0	0	300	0	0	0	0
(e) Chained hashing	0	0	0	7	293	0	0	0
(f) Matrix	0	1	0	0	5	294	0	0
(g) Randomly Accessed Vector	0	0	0	0	3	0	279	18
(h) Sequentially Accessed Vector	0	0	0	0	0	0	44	256

Table 1: Accuracy of SVM on a symbolic trace and natural feature evaluation

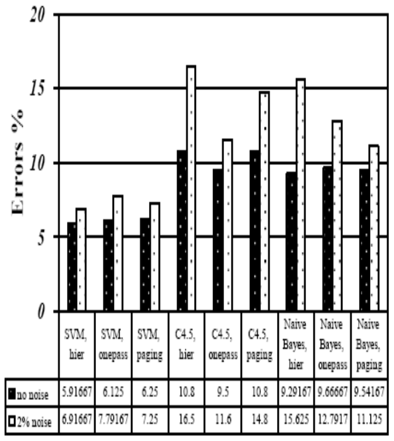
of the errors occur between vectors with a random access pattern and vectors with a sequential access pattern. The difference between definitions of these two data structures is indeed fuzzy.

6.4 Robustness

In the current research we addressed the case where the program uses a single data structure. In the real world accesses to a number of different data structures are interleaved within a single program. It is important to examine the robustness of our technique to the addition of noise. In Figure 3 we compare the robustness per classifier and per clustering algorithm. As it can be seen, all



	SVM, hier	SVM, onepass	SVM, paging	C4.5, hier	C4.5, onepass	C4.5, paging	Naive Bayes, hier	Naive Bayes, onepass	Naive Bayes, paging
■ no noise	7.58333	7	6.95833	11.4	10.1	9.2	16.5833	15.0833	16.75
□ 2% noise	8.41667	7.95833	7.75	13	12	11.2	18.125	12.0417	11.9167



	SVM, hier	SVM, onepass	SVM, paging	C4.5, hier	C4.5, onepass	C4.5, paging	Naive Bayes, hier	Naive Bayes, onepass	Naive Bayes, paging
■ no noise	5.91667	6.125	6.25	10.8	9.5	10.8	9.29167	9.66667	9.54167
□ 2% noise	6.91667	7.79167	7.25	16.5	11.6	14.8	15.625	12.7917	11.125

Fig. 3: The robustness per classifier and per clustering algorithm using the clusters0to3 subset of clustering levels.
On the left the natural feature evaluation; on the right the binary one.

(classifier, clustering algorithm) pairs are very sensitive to noise.

6.5 Statistical Significance of the Results

A statistical analysis of the results shows that with over 95% confidence the error is under 1%.

7 Concluding Remarks

In this study we have described the machine learning approach to classifying any subset of data structures. We have considered the case where the program uses a single data structure, which only supports searches. Scaling this technique to data structures, which support insertions/deletions, and to the case of multiple data structures is left for further studies. Our technique is easily extendable to additional data structures and implementations.

Our statistical measurements show the high quality of classification in the absence of noise, especially when using the SVM and C4.5 classifiers. On the other hand, the approach to feature selection described in our solution is very sensitive to noise. We will have to overcome this lack of robustness in order to proceed to real world applications with accesses to a number of different data structures interleaved within a single program.

When we used the C4.5 classifier we got a classification algorithm (decision tree) for every subset of data structures. The time required to classify the trace of memory accesses into a data structure is $O(t)$, where t is the total time to calculate all the features appearing on the appropriate path in the tree. Thus we considered techniques to reduce the time spent on the calculation of a single feature without the accuracy degradation, such as:

- The usage of the quick paging clustering model.
- The usage of the binary feature evaluation, instead of the natural one (in this case we need not to search the entire trace).

Let us conclude with directions for the future research:

(a) At the current stage we examined the case of static data structures, i.e., only searches are supported (no insertions/deletions). To scale up this technique to dynamic data structures we need also to consider the type of the memory access (read or write). We can add this information as an additional entry to our tuple.

(b) In general, memory accesses may be generated by a number of sources. Moreover, sometimes several data structures are processed simultaneously (e.g., matrix multiplication).

    i) Sometimes our techniques can still be applied if we change the definition of the data structure. For example, we may consider a pair of multiplied matrices as a single data structure.

ii) If several data structures are processed sequentially we may also consider contributions of both time and code locality.

(c) The robustness of our approach has to be improved in order to proceed to real world applications that access a number of different data structures interleaved within a single program.

(d) In addition we should take into account the time required to calculate the features. Thus we could gain both accuracy and classification time by better feature reduction.

References

[1] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press and McGraw-Hill Book Company, 1989.

[2] K. Ebcioglu, E. Altman, and E. Hokenek. A Java ILP machine based on fast dynamic compilation. In *IEEE MASCOTS International Workshop on Security and Efficiency Aspects of Java*, 1997.

[3] EMC Corporation. *Symmetrix 3000 and 5000 Enterprise Storage Systems Product Description Guide*, 1999.

[4] S.C. Johnson. Hierarchical clustering schemes. In *Psychometrika*, volume 32, pages 241–254, 1967.

[5] T. Mitchell. *Machine Learning.* McGraw Hill, 1997.

[6] J. Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1993.

[7] Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[8] S. Rieber and U. P. Marathe. The single pass clustering method. Technical Report ISR-16, Cornell University, the Department of Computer Science, 1969. to the NDF.

[9] Gerard Salton and Michael McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill Book Company, 1984.

[10] R.R. Sokal and C.D. Michener. A statistical method for evaluating systematic relationships. *The University of Kansas Scientific Bulletin*, 38:1409–1438, 1958.

[11] Vladimir Vapnik. *The Nature of Statistical Learning Theory.* Springer-Verlag, New York, 1995.