Learning Nested Concept Classes with Limited Storage*

David Heath, Simon Kasif, S. Rao Kosaraju, Steven Salzberg, and Gregory Sullivan
Department of Computer Science
The Johns Hopkins University
Baltimore, MD 21218

Abstract

Many existing learning methods use incremental algorithms that construct a generalization in one pass through a set of training data and modify it in subsequent passes (e.g., perceptrons, neural nets, and decision trees). Most of these methods do not store the entire training set, in essence employing a limited storage requirement that abstracts the notion of a compressed representation. The question we address is, how much additional processing time is required for methods with limited storage? Processing time for learning algorithms is equated in this paper with the number of passes necessary through a data set to obtain a correct generalization. For instance, neural nets require many passes through a data set before converging. Decision trees require fewer passes, but precise bounds are unknown.

We consider limited storage algorithms for a particular concept class, nested hyperrectangles. We prove bounds that illustrate the fundamental trade-off between storage requirements and processing time required to learn an optimal structure. It turns out that our lower bounds apply to other algorithms and concept classes (e.g., decision trees) as well. Notably, imposing storage limitations on the learning task forces one to devise a completely different algorithm to reduce the number of passes. We also briefly discuss parallel learning algorithms.

1 Introduction

Many existing learning methods attempt to create concise generalizations from a set of examples. Besides saving storage, small generalizations are easier to summarize and communicate to others. A common learning method will construct a generalization after one pass through a set of training data, and modify it in subsequent passes to make it smaller or more accurate. Perceptron methods, neural nets, and decision tree techniques all fit this paradigm. Most of these methods do not store the entire set of training data. When processing is completed, the only thing they store is a generalized data structure such as a tree, a matrix of weights, or a set of geometric clusters. The issue of limiting the storage of a learning algorithm is one abstraction of the notion of a compressed representation. The question we address is how many passes through a data set are required to obtain a "correct" (i.e., accurate but minimum in size) generalization if we are only allowed to store the generalization. This issue is equivalent to analyzing an algorithm that has a limited storage requirement.

Fixed storage is an important consideration for several reasons. First of all, some learning models *always* use fixed storage. Neural net learning algorithms, for example, have a fixed number of nodes and edges, and only change the weights on the edges. Decision tree algorithms can in principle grow without bound, but in practice researchers have devised many techniques for restricting their growth [Quinlan 1986, Utgoff 1989]. Instance-based techniques such as those of Salzberg [1989ab] and Aha and Kibler [1989] attempt to store as few examples as possible in order to minimize storage. Second, fixed storage is a realistic constraint from the perspective of cognitive modelling - human learning behavior clearly must adhere to some storage limitations. Finally, there is experimental evidence that restricting storage actually leads to better performance, especially if the input data is noisy [Aha and Kibler 1989, Quinlan 1986]. The intuition behind this result is that by throwing away noisy data, an algorithm can construct a more accurate generalization.

Our results show that if a fixed-storage algorithm attempts to create a simple concept structure, then it *cannot* generalize on-line without losing accuracy By "simple" we mean a generalization that is both minimum in size and an accurate model of the data; i.e., it classifies all the training examples correctly. We also show that by making a number of additional passes (depending on the number of concepts) through the data set, an algorithm can create an optimal structure. Our main goal is to demonstrate that there exists a fundamental trade-off between the storage available and the number of passes required for learning an optimal structure. There are few results comparable to ours on the number of passes re-

*This research supported in part by Air Force Office of Scientific Research under Grant AFOSR-89-1151, National Science Foundation under Grant IRI-88-09324 and NSF/DARPA under Grant CCR-8908092.

[1] Supported by NSF under grant CCR-8804284 and NSF/DARPA under grant CCR-8808092.

quired to learn a concept. Typical theory results, rather, give estimates of the size of the input data set, not of the number of presentations required. This work is an important step towards formalizing the capabilities of incremental vs. non-incremental learning algorithms. Any algorithm that does not save all training examples is to some extent incremental, since upon presentation of new inputs the algorithm cannot re-compute a generalization using all previous examples.

The learning framework considered in this paper is computing generalizations in the form of geometric concept classes. Many important learning algorithms fall in this category, e.g., perceptron learning [Rosenblatt 1959], instance-based learning [Aha and Kibler 1989], decision tree models [Quinlan 1986] and hyperrectangles [Salzberg 1990]. We focus on the concept class defined by nested hyperrectangles, although many of our results are applicable to other concept classes. In fact, our impossibility results apply to any convex partitioning of feature space, such as decision trees and perceptrons.

The learning model we consider has fixed number of storage locations; i.e., it is not permitted to store and process all training examples at once. The limited storage requirement is applicable only during the learning (or training) phase. That is, our algorithms must operate incrementally, storing a limited number of intermediate results during each pass through a data set and modifying the partially constructed generalization in subsequent passes. For both cognitive and practical reasons, much experimental learning research has focused on the development of incremental models [Utgoff 1989].

2 Nested Hyperrectangles

Recent experimental research on learning from examples has shown that concepts in the shape of hyperrectangles are a useful generalization in a variety of real-world domains [Salzberg 1989ab, 1990]. In this work, rectangular-shaped generalizations are created from the training examples, and are then used for classification. Rectangles may be nested inside one another to arbitrary depth, and new examples are classified by the innermost rectangle containing them. Thus nested rectangles may be thought of as *exceptions* to the surrounding rectangles. This learning model is called the Nested Generalized Exemplar (NGE) model. Experimental results with this model thus far have shown that it compares very favorably with several other models, including decision trees, rule-based methods, statistical techniques, and neural nets [Salzberg 89ab].

Independently of the experimental work cited above, Helmbold, Sloan, and Warmuth [1989] have produced very promising theoretical results for nested rectangular concepts. In particular, they have developed a learning algorithm for binary classification problems that creates strictly nested rectangles, and that makes predictions about new examples on-line. They have proven that their algorithm is optimal with respect to several criteria, including the probability that the algorithm produces a hypothesis with small error [Valiant 1984] and the expected total number of mistakes for classification of the first t examples. The algorithm applies to all intersection closed classes, which include orthogonal rectangles in R^n, monomials, and other concepts. The main assumption behind their model is that it must be possible to classify the training examples with strictly nested rectangles.

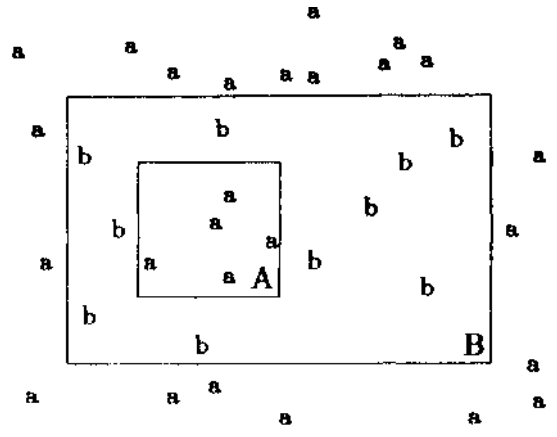

Figure 1: Categorizing points using rectangles

Given that the learning community has found nested rectangles a useful concept class, and that the theoretical community has proven some further results about this same concept class, we have been led to investigate the ability of an algorithm to construct an optimal number of nested rectangles given only limited storage. We will argue below that our results apply to other well-known concept classes, including the partitionings induced by decision trees and perceptrons.

3 Preliminaries

An *example* is defined simply as a vector of real-valued numbers, plus a category label. For instance, we may be considering a problem where medical patients are represented by a set of real numbers including heart rate, blood pressure, etc., and our task is to categorize the patients as "in-patient" or "out-patient." For our purposes, an example is just a point in Euclidean space, where the dimensionality of the space is determined by the number of attributes measured for each example. We will categorize points by using axis-parallel hyperrectangles, where each rectangle R_i is labeled with a category $C(R_i)$ such as "out-patient." A point is categorized by the innermost rectangle containing it. Figure 1 illustrates how categories are assigned. In the figure, lowercase letters indicate points belonging to categories a and b, and uppercase letter indicate the category labels of the rectangles. Notice that points not contained by any rectangle are assigned to category A, which corresponds to a *default* category. Only two rectangles are required to classify all the points in Figure 1.

The general problem definition is as follows: we are given n points in a d-dimensional space, and we are asked to construct a minimum set of *strictly* nested hyperrectangles that will correctly classify the set. We will assume that each point belongs to one of two classes (i.e., we have a binary classification problem).

Our algorithms learn by processing examples one at a time, in a random order. The algorithm is allowed to store no more than a fixed number *S* of the examples. In addition, the algorithm may have some additional constant amount of storage. On each *pass* through the data, the algorithm sees all of the examples exactly once. In most cases, the order of the examples on each pass is independent of the order on other passes.

Given these definitions, we would like to answer the following general question: how many passes *p* through the data are required to construct a minimum set of nested hyperrectangles? We will present several algorithms, and show how the number of passes required changes as a function of the amount of storage 5 and of the minimum number of rectangles R.

4 Learnability

There are some input sets which cannot be learned with nested isothetic rectangles. We say a set of examples is *learnable* if the nested rectangle problem for this set has a solution. Here we present a necessary and sufficient condition for the learnability of a set of examples.

Definition 1: In a binary *classification problem, an isothetic hypeerectangle with nonzero area is called* a blocking rectangle *if every edge of the hyperrectangle intersects points of both classes.*

Theorem 4.1: *A set of points from two classes is learnable if and only if there does not exist a blocking rectangle for the set.*

A proof may be found in Heath *et al.,* [90].

5 Static Algorithm

When all of the examples can be stored in the memory (5 ≥ n), the nested rectangle problem can be solved by a modified plane sweep. We make one pass through the examples, during which we store all examples.

We sweep *2d* orthonormal hyperplanes, two along each axis. Each hyperplane defines two halfspaces: *inside* and *outside.* The intersection of the *2d inside* halfspaces is a hyperrectangle. Initially we position the hyperplanes so that the hyperrectangle, Ro, is the smallest that contains all of input points. Let the category of Ro (which is not a partitioning rectangle) be $C(R_0)$.

The computation proceeds in *R* steps. In each step, we find the next hyperrectangle, R_i+i, nested inside Ri, by sweeping each hyperplane inward (towards the *inside* halfspace it defines) until it intersects a point not belonging to category $C(R_i)$. In some steps, some hyperplanes may not move at all. The new positions of the hyperplanes define the hyperrectangle R_{i+1}, which is output.

Note that this algorithm requires that the points are sorted along all dimensions, which requires O(dn log n) time. Each pair of hyperplanes sweeps over *n* points, so the sweep takes $O(dn)$ time. Thus, the total time needed, including that for sorting, is $O(dn \log n)$.

Theorem 5,1: Given *n points in d-dimensional space, the nested rectangle problem can be solved in* $O(dn \log n)$ *time and O(dn) space.*

6 Limited Memory Algorithms

6.1 Simple limited memory algorithm

The static algorithm can easily be converted to run with limited memory. First, suppose we have fixed memory sufficient to store all rectangles, but not all examples. We show that *R* passes will suffice to find *R* rectangles.

Intuitively, to find the outermost rectangle, our algorithm finds the maximum and the minimum point in each dimension. These *2d* points define the edges of the first hyperrectangle. Inductively, assume *i* rectangles have been constructed and the category of R_i is $C(R_i)$. Let *inside(Ri)* be the set of points inside the i^{th} rectangle. We now find the maximum and minimum point in each dimension for category $C(R_i + 1)$ that belong to *inside(Ri).*

This can be done in one pass by storing, for each dimension, the smallest and largest point of category $C(R_i + 1)$ in *inside*(R_i) seen so far. We revise our current estimate of the smallest and largest points (if necessary) and continue as each new point is processed.

Since we can find the next partitioning hyperrectangle in one pass, we can solve the partitioning problem with 4d+1 storage locations in *R* passes.

Theorem 6.1: *Given* 4d+1 *storage locations, it is possible to solve the d-dimensional nested rectangle problem in R passes, where R is the number of rectangles.*

This algorithm can be seen as a line adjustment algorithm similar to perceptrons. Unlike the standard perception algorithm, it has the following characteristics.

- It is guaranteed to converge in *R* passes.
- Hyperplanes always move in the same direction.
- A hyperplane makes large incremental adjustments towards its final location.
- The network can classify input data that is not classifiable using the standard perceptron algorithm
- A hyperplane defining rectangle R_i+1will not be adjusted until R_i has reached its final location.

Our major results, described below, illustrate that in many cases we can improve the performance of our algorithm to learn the concept structure correctly in far fewer passes. Additionally, if *R* is small with respect to n, then *R* passes are *required* to define the rectangles.

6.2 Speeding up the limited memory algorithm

In this section we show that with a simple modification of the static algorithm, we can design an algorithm for the nested rectangle problem that runs in fewer than *R* passes, where *R* is the number of rectangles A more efficient scheme will be described in the next section.

As above, the outermost rectangle can be found in one pass with only 2d+1 storage locations. Assume inductively that *i* rectangles have been found using storage *S* = 2d(2 + *S)* + 1. We use *2d* storage locations for maintaining a hyperrectangle *W*, which is a window outside of which we have found all rectangles *{R*$_1$*,* .. , *Ri},* where *Ri* is innermost. Initially, *W* contains all the examples (no rectangles have been found). All points outside *W* can be ignored while the algorithm positions one or more

rectangles within *R{.* For each of the *2d* hyperplanes that define *W,* we allot *s* memory locations, which will be used to find the *s* closest points (of any color) to the hyperplane that are inside *W.* All of these points can be found in one pass. Once they are in memory, the set of s points associated with each line is sorted.

We define an *alternation* to be a pair of points that belong to different categories and are adjacent in a sorted list. The alternations in each list can be found, and then matched up to find partitioning rectangles. If each list has at least one alternation, the outermost alternations in each list define a partitioning rectangle Ri + 1- Every point stored that is not inside *Ri + 1* is removed from the lists, since these points cannot define rectangles nested within R_{i+1}. Now, the outermost alternations in each list define R_{i+2}- We continue placing rectangles this way until at least one list has no alternations. Note that some list could have no alternations at the beginning of the pass. In this case, we simply find no rectangles in that pass. In any case, we redefine *W* as follows: for each list that still contains alternations, we move its associated hyperplane to the last alternation that was deleted (if any). We move every other hyperplane to the innermost point in its list. Because these lists have no alternations, they cannot generate partitioning rectangles. Since at least one list has no alternations, at least one hyperplane has moved in by s points (i.e., at least *s* points have been excluded from *W).* Since no point that leaves *W* can reenter, *W* will be empty in no more than *n/s* passes.

Theorem 6.2: *Given S storage locations, it is possible to solve the d-dimensional nested rectangle problem in 0(nd/S) passes.*

In particular, when the number of rectangles *R* is larger than $O(\sqrt{nd})$, and $S = \theta(R)$, the number of passes is smaller than *R.* In other words, we can always solve the problem in $O(\sqrt{nd})$ passes irrespective of the number of rectangles.

6.3 Limited memory with sampling

All of the preceding algorithms work by finding alternations from the outside, working inwards. In this section we present a different algorithm that works in one dimension. This algorithm finds alternations on the real line, so is applicable to any convex partitioning of the line, such as decision trees and intervals. The algorithm is fairly complex and demonstrates the difficulty of learning efficiently with limited storage even in one dimension.

As before, we define an alternation to be a pair of adjacent points which belong to different categories. Note that once we detect all the alternations it is easy to find the concept structure. Clearly, if the concept being acquired is classifiable with *R* rectangles, then there are no more than *2R* alternations. Our algorithm will detect and store all the alternations. The intuitive explanation is as follows. We use one pass to find the size of the input space, and then we partition the input space into intervals of approximately equal size. The algorithm maintains markers for approximately / intervals, where *I* is chosen appropriately (see below). Once these intervals are established, we can find the outermost two alternations (if they exist) in each interval in one pass. If an interval contains no alternations, the input points in the interval are ignored in subsequent passes. Our algorithm keeps track of only the *active* intervals, namely intervals that have alternations in them.

space available	*passes needed*	*I*
$S \leq \frac{R}{\log^3 n}$	$O(\frac{R}{S})$	S
$\frac{R}{\log^3 n} \leq S \leq R \log n$	$O(\log^{3/2} n \sqrt{R/S})$	$\frac{\sqrt{RS}}{\log^{3/2} n}$
$R \log n = S$	$O(\log n)$	$\frac{R}{\log n}$

Table 1: Choosing the number of intervals *I*

The following schematic, code fragment is executed repeatedly until all alternations have been found.

1. while $\|Int\| < I$ and some $i \in Int$ is splittable
2. split all $i \in Int$
3. check each interval for an alternation
4. discard intervals not containing alternations
5. End while
6. Detect one or two alternations from each *i E Int*

Int. is the current set of intervals, initially the entire real line, containing all n of the input points. The method we use to split the intervals in step 2 is guaranteed to split each interval into a small constant number of subintervals, each of which contains no more than a constant fraction of the interval's points. Since the intervals will be shrinking in size by a constant fraction each time step 2 is executed, the while loop will be iterated no more than *()(\log n)* times. Let $s = S/I$ be the storage available per interval, where $1 \leq s \leq \log^2 n$.

The while loop guarantees that there are at least *I* intervals, and each one contains an alternation. Step (6) will find *I* alternations in one pass, so will require at most *2R/I* passes. *O(I)* storage locations will be used to maintain the intervals.

In step 2, we make use of a limited storage splitting technique developed by Munro and Paterson [80]. It splits each interval into three subintervals, each of which is no larger than a constant fraction of the size of the interval. To split / intervals, it takes $O(\log^2 n/s)$ passes, using *sI* storage locations, where $1 \leq s \leq \log^2 n$. (If $s =$ o(log n), the examples must appear in the same order for the $0(\log^2 n/s)$ passes). Since step 2 is executed at most log n times, a total of $O(\log^3 n/s)$ passes will be needed for step 2. The algorithm uses a total of $(2R/I +$ log *n/s)* passes with $S = sI$ memory locations.

The choice of/ depends on *R* and the memory/speed trade-off desired (see Table 1). In practice, we always have enough space for the rectangles $1(S \geq R)$ and therefore we can find all the alternations in $O(\log^{3/2} n)$ passes.

There are many situations in which the number of rectangles *R* is not known beforehand. In this case, a binary search technique can be applied to the above algorithm. Initially, we run the algorithm, assuming that 2 rectan-

gles are sufficient to completely learn the structure. In one pass, it is possible to verify that the solution generated does indeed correctly classify the input. If not, we can double the number of rectangles and try again. We continue doubling the number of rectangles until the algorithm successfully finds the nested rectangles, using *O(R)* storage. Because the number of rectangles is doubled in each step, and successful classification is guaranteed when the number of rectangles is at least *R,* the partitioning algorithm will run at most [log *K]* times. Thus, the total number of passes needed to determine *R* and find the rectangles is $O(\log^{3/2} n \log R)$.

A generalization of the algorithm to higher dimensions is not obvious and is being investigated. We are considering a randomized variant of this algorithm for multiple dimensions. It partitions the input set into regions, each containing a fixed fraction of the points. Then the algorithm finds a rectangle in each region in each pass. While such an algorithm may work well in practice, it may not find the minimum set of rectangles, and may fail to find a correct generalization when one exists.

7 Lower Bound

In this section we show that when the concept structure is fairly simple and the amount of storage necessary to represent the concept is therefore small, the number of passes required is proportional to the number of alternations. Intuitively, the alternations represent the concept boundaries, so the size of the representation constructed by many learning algorithms is proportional to the number of alternations in the input set. For example, decision tree learning algorithms must construct one branch for every alternation. The theorem below is stated in terms of rectangles but our lower bound technique also applies to the number of passes necessary to find alternations. The same result holds for decision trees or any other method of partitioning into convex regions.

We will use a comparison based model which is sufficiently strong to support our previous algorithm. The following theorem shows that any comparison-based algorithm that has *kR* storage, where *k* is some constant needs at least *R* passes of the examples when *R* is sufficiently small compared to n $(\text{i.e., } n \geq (2kR+1)!)$.

Theorem 7.1: *Any comparison-based algorithm for solving the nested rectangle problem with k R storage requires R passes to find R partitioning rectangles, when there are* $n \geq (2kR+1)!$ *points in the input,*

A full proof of this theorem may be found in Heath *et* al., [90]. We give an intuitive explanation for the case when *R* = 2, where two passes are needed to solve the nested rectangle problem when $S \leq n/2$. We will find, for any algorithm, two distinct sets of inputs indistinguishable by the algorithm that cannot be categorized with the same set of two nested rectangles. Suppose we have two categories, "blue" and "green." Consider a trainer that presents points in the blue category first. Because the learner has limited storage, it will eventually forget at least one blue point. After all n/2 blue points are presented, the trainer presents three points from the green category, surrounding the forgotten blue point. It can choose either of two ways to position the green points (see Figure 2). The algorithm will have the opportunity to find the outermost (green) partitioning rectangle. To correctly place the inner (blue) rectangle, the program must surround the forgotten blue point. However, the program will not be able to tell if the input is given by Case 1 or Case 2 of Figure 2. If the program were to decide one way, the trainer could have chosen to present the other case. Because no comparisons between the forgotten blue points and the three green points are performed, the two cases are indistinguishable.

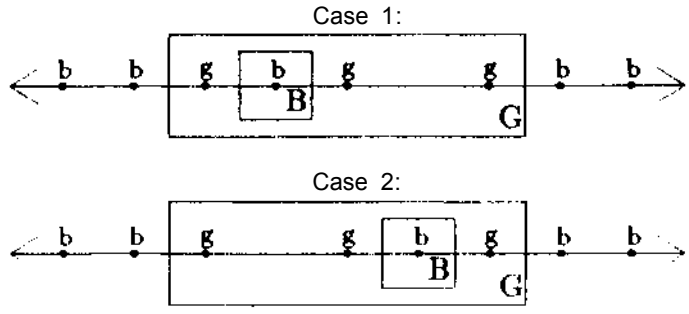

Figure 2: Two possible input patterns

8 Parallel Algorithms

Connectionist architectures have natural parallel implementations. This has motivated us to examine whether our algorithms also have natural parallel implementations. Given *n* points on the line, we can solve the nested rectangle problem in $O(n/P)$ time using *P* processors, where $P \leq n/\log n$. For details, see Heath *et* a/., [90].

Parallel algorithms can be developed to solve the nested rectangle problem in any fixed number of dimensions. One would hope that our parallel algorithm could be generalized to work in any number of dimensions. However, we show that when the dimensionality is not fixed, the nested rectangle problem becomes log-space complete for P, or *P-complete.* This suggests that when the dimensionality of the problem is not fixed, there is no efficient (i.e. polylog time with polynomial number of processors) parallel algorithm. This condition is usually considered as a strong indication of inherent sequentiality, since there are no known methods to achieve significant speed-ups of such problems on parallel architectures. See Heath *et* al., [90], for a proof of this result.

9 Conclusion

Intuitively, a learning program with fixed storage cannot create an optimal set of nested rectangles to classify a set of points (examples). The problem is that, since memory is limited, the program *must* forget some of the examples, and these examples may be misclassified as a result. It is clear, however, that a partial generalization constructed in *p* passes can be further refined using more passes. This paper makes precise the trade-offs between storage, number of passes, and classification accuracy.

Number of rectangles	*Lower bound*	*Upper bound*
$(2kR+1)! \leq n$	R passes	R passes
$(2kR+1)! > n,\ R \leq c\log^{3/2} n$	None	R passes
$R \geq c\log^{3/2} n$	None	$O(\log^{3/2} n)$ passes
$R \geq n/\log^{3/2} n$	None	$O(n/R)$ passes

Table 2: Number of passes required as R increases

When an algorithm only has enough memory to store the *R* rectangles themselves, it requires (in the worst case) no more than *R* passes through the data. In addition, when *R* is large with respect to the number of examples n, we can do much better. For instance, when $R \geq c\log^{3/2} n$, for some constant r, we only need $O(\log^{3/2} n)$ passes through the data. For most learning problems, we do not know the size of *R* in advance (i.e., we do not know how compact the generalization might be). However, our algorithm can still learn both *R* and the target concept accurately in $O(\log R \log^{3/2} n)$ passes. Table 2 summarizes our results for different ranges of *R*.

From a practical standpoint, these results allow one to make some statements about the trade-off between processing time and storage for algorithms that learn nested hyperrectangles. As long as storage is not limited, we can use an algorithm that (with one pass through the data) runs in $O(dn\log n)$ time, where *d* is the number of features for each example, to find an optimal (i.e., minimum) set of rectangles. If storage is fixed and the number of examples is large, then $\min(R, O(\log^{3/2} n), n/S)$ passes are sufficient to find the optimal set of nested rectangles, depending on the size of *R* with respect to *n*. If fewer passes are allowed, then the rectangular concepts learned by the algorithm may misclassify some of the examples.

We have studied the complexity of non-incremental learning algorithms for parallel models of computation. In Heath *et al.* [1990], we present an optimal parallel algorithm for learning in one dimension, and show that learning nested hyperrectangles when the dimensionality is not fixed is P-complete (inherently sequential).

The major open problems we are currently considering include developing efficient limited memory algorithms for data sets with multiple dimensions. We plan to implement these algorithms for experimental tests on real data. We also are working on improvements to our currently crude parallel algorithms for multiple dimensions. Additionally, we are considering the problem of efficiently constructing near-optimal concept structures.

References

[Aha and Kibler, 1989] D. Aha and D. Kibler. Noise-tolerant instance-based learning algorithms. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence,* Detroit, MI, 1989. Morgan Kaufmann

[Heath *et al.,* 1990] D. Heath, S. Kasif, R. Kosaraju, S. Salzberg, and G. Sullivan. Learning nested concept classes with limited storage. Technical Report JHU-90/9, Computer Science Department, The Johns Hopkins University, 1990.

iHelmbold *et al,* 1989] D. Helmbold, R. Sloan, and M. Warmuth. Learning nested differences of intersection closed concept classes. In *Proceedings of the 1989 Workshop on Computational Learning Theory,* San Mateo, 1989.

[Karp and Ramachandran, 1988] R. Karp and V. Ramachandran. A survey of parallel algorithms for shared-memory machines. Technical Report UOB/CSD 88/408, Computer Science Division, University of California, Berkeley, 1988.

[Munro and Paterson, 1980] J. Munro and M. Paterson. Selection and sorting with limited storage. *Theoretical Computer Science,* 12(3):315- 323, 1980.

[Quinlan, 1986] JR. Quinlan. Induction of decision trees. *Machine Learning,* 1(1):81- 106, 1986.

[Salzberg, 1989a] S. Salzberg. *Learning with Nested Generalized Exemplars.* PhD thesis, Harvard Universify, 1989. Technical Report TR-14-89.

[Salzberg, 1989b] S. Salzberg. Nested hyper-rectangles for exemplar-based learning. In K.P. Jantke, editor, *Analogical and Inductive Inference: International Workshop ATI '89,* pages 184 201. Springer-Verlag, Berlin, 1989.

[Salzberg, 1991] S. Salzberg. A nested hyperrectangle learning method. *Machine Learning,* 6:251-276, 1991.

[Utgoff, 1989] P. Utgoff. Incremental induction of decision trees. *Machine Learning,* 4(2):161-186, 1989.

[Valiant, 1984] L. Valiant. A theory of the learnable. *Communications of the ACM,* 27(11): 1134-1142, 1984.