Bisecting the Version Space

Anish K. Arora
Daniel L. Dvorak
Thomas C. Vinson

AI TR87-62 **August, 1987**

This research was conducted by the authors in the *Machine Learning* course of Spring 1987, taught by Professor Bruce Porter.

Bisecting the Version Space

Anish K. Arora
Daniel L. Dvorak
Thomas C. Vinson

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

August 12, 1987

Abstract

A class of problem in machine learning is concept learning where an experimenter poses questions to an oracle. The *ideal experimenter* learns the target concept with the fewest questions. For learners using Mitchell's version space algorithm (*a.k.a.* candidate elimination algorithm) for concept formation, Mitchell had conjectured that the best questions would be those that bisect the version space. To date, no published research has attempted to verify the conjecture (or even define what it means to "bisect" the version space).

This paper reports the results of an effort to devise an ideal experimenter that has no *a priori* knowledge about the target concept. Several strategies were programmed and tested on single- and multiple-attribute learning on concept hierarchy trees and directed acyclic graphs. We obtained the best results with an experimenter that:

- selects the training instance based on a concept midway on a path between elements of the G and S sets bounding the version space, and
- for multiple-attribute concepts it modifies only one attribute at a time (divide-and-conquer).

For the special case where training instances are allowed to be non-leaf nodes in the concept hierarchy, an even better result can be obtained with the addition of a new rule for updating the version space.

Contents

1 Introduction 3

2 Learning from Experimentation 3

3 Learning in the Version Space 4

4 Experiment Generation Strategies 5

4.1 genrandom 5

4.2 gen1 6

4.3 gen0 7

4.4 gen 7

4.5 genbisect 8

5 Performance and Bias 9

6 Directed Acyclic Graphs 10

6.1 geninterior 11

6.2 DAG Performance Results 11

7 Multiple-Attribute Concepts 13

8 Conclusions 14

9 Acknowledgements 14

10 References 15

A Proof of Optimality 16

B Software 18

1 Introduction

In the field of machine learning there is a class of systems that perform "learning from experimentation" where a learner acquires a theory or concept by posing questions to an oracle. The oracle can be a human or a program or a mechanism. The technique is suited for the situations where previous knowledge is not sufficient to explain some observations — where reasoning has to be replaced by experimentation. Examples of such systems are Meta-DENDRAL [4], LEX [1], MARVIN [3] and EG [2]. Standard assumptions in these systems are that:

- the oracle gives correct answers (no noise),
- the learner knows the theory representation language,
- the learner is given an initial positive training instance, and
- *the learner tries to ask the fewest questions.*

This last point is the subject of this paper. At every stage in trying to learn a concept, the experimenter must try to find the *best* question to ask the oracle.

This paper first quickly reviews the general model of learning from experimentation and the version space algorithm. It then describes several experiment generation strategies that we devised and tested. Experimental results are shown for single-attribute and multiple-attribute concepts in concept hierarchy *trees* and *graphs*.

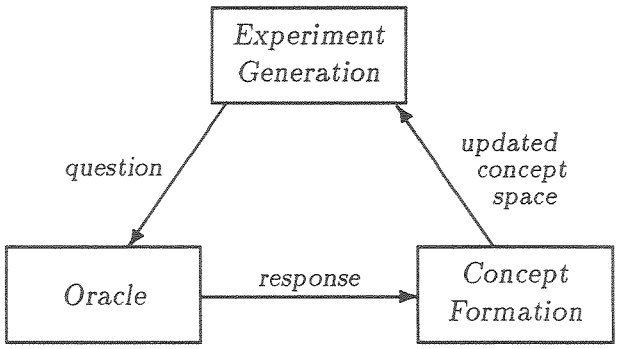

Figure 1: The learning cycle

2 Learning from Experimentation

The process of learning from experimentation is a continuous cycle in which the learner asks a question of the oracle, evaluates the oracle's response, and then asks a new question. The cycle continues until the learner narrows down the space of possible concepts to one concept.

The learner performs two tasks in this scenario — *experiment generation* and *concept formation* — and there is a continuous interplay between the two. Partial theories guide experiment generation and experiment results guide concept formation.

Concept formation may be based upon any of several techniques — clustering, induction, explanation-based learning, analogy, etc. The design of the experiment generator is necessarily influenced by the concept formation techniques. In this research we have selected Mitchell's version space algorithm (candidate elimination algorithm) [5] as the concept formation technique. Our goal is to create the "ideal" experiment generator — the one that asks the fewest questions in the learning cycle.

3 Learning in the Version Space

Concepts are represented as nodes in an acyclic concept hierarchy graph with directed arcs representing a *more-specific-than* relation between pairs of nodes. The version space algorithm performs a bi-directional search in this concept hierarchy graph that progressively narrows the distance between the two "boundaries" of the search space. These boundaries, named G and S, are defined as follows:

G = the most general set of generalizations consistent with the observed instances;

S = the most specific set of generalizations consistent with the observed instances.

Initially, G is set equal to the root of the concept hierarchy graph and S is set equal to the first positive training instance. With subsequent positive and negative training instances, the algorithm updates G and S in accord with their definitions. The target concept always lies somewhere in the space between the two boundaries (the so-called *version space*). Eventually, the G and S boundaries meet at the target concept and the learning algorithm terminates.

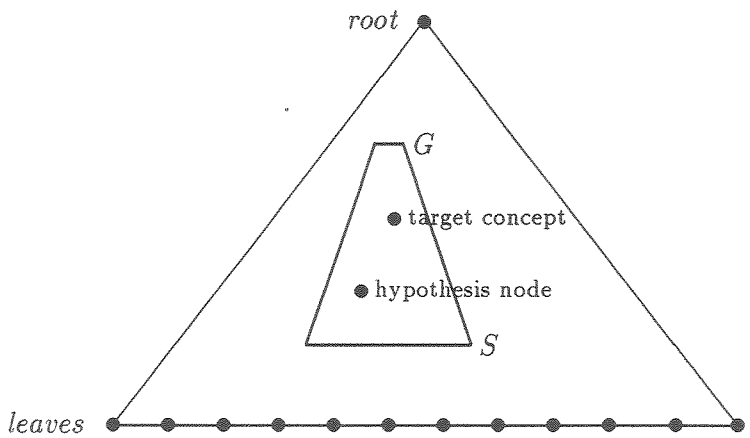

Figure 2: Positions inside a concept hierarchy tree

Figure 2 shows the relative positions of different items as learning proceeds in a concept hierarchy tree. G begins at the root and descends as it is specialized by negative training instances. S begins as a leaf and ascends as it is generalized by positive training instances. The *target concept* is always somewhere in between and eventually, after enough training instances, G and S converge to the target concept. Traditionally, training instances are leaf nodes. A smart learner selects each training instance to test some hypothesis about the target

concept. Thus, we introduce the notion of a *hypothesis node* to represent the concept that the learner is actually asking about indirectly through a leaf node. In one of the experiment generators described later, we part with tradition and allow the learner to directly ask the oracle about a hypothesis node.

4 Experiment Generation Strategies

For any experiment generator to be considered optimal, it must ask questions that cause G and S to converge rapidly. By implication, it must avoid "dumb" questions. Any question that is *not* covered by G or *is* covered by S is a "dumb" question since it can be answered without asking the oracle. Such questions cause no narrowing of the version space. All other questions are termed "intelligent", and the best of these maximize the change in G or S.

There are many possible strategies for selecting intelligent questions. These strategies will vary according to bias, computational complexity, and amount of domain-specific knowledge. Mitchell has conjectured that the ideal strategy would be one that, in some sense, "bisects" the version space with each question. To date, no published research has tested and confirmed that hypothesis. However, some unpublished results have suggested that it is not possible to do better than random questioning. (Indeed, random questioning works very well with the version space algorithm). This research seeks an experiment generation algorithm that is optimal in the number of questions asked and is a domain-independent, computationally-modest strategy.

We devised and tested several experiment generators. The sole figure of merit for comparison purposes is the number of questions asked while learning a concept. Figure 3 shows the performance of each generator on a symmetric 9-level 511-node[1] concept hierarchy tree having a branching factor of 2. In the following subsections we describe the strategy used by each generator and explain its resulting performance.

4.1 genrandom

This generator randomly selects a leaf node descended from G. That node may also be descended from S, which would be a "dumb" question. However, we have eliminated such questions in measuring its performance since our objective with `genrandom` was to establish as a benchmark the performance of an "intelligent" but random questioner.

The strategy of questioning at random is actually a reasonable one since, at any moment, all that is known about the target concept has been summarized in G and S. Thus, on the assumption that "one guess is as good as another", this generator selects a hypothesis node randomly from the version space. Any such guess, regardless of whether it is classified by the

[1] A 511-node 9-level tree was chosen for two reasons. First, a 9-node path from S to G can be repeatedly bisected yielding a true middle node each time (the next larger tree having this property is a 17-level tree of 131,071 nodes). The second reason for the 511-node tree was time; the execution time for our experiment generators averaged 3–4 hours when run through all 511 nodes, which was necessary when we later converted the tree to a graph.

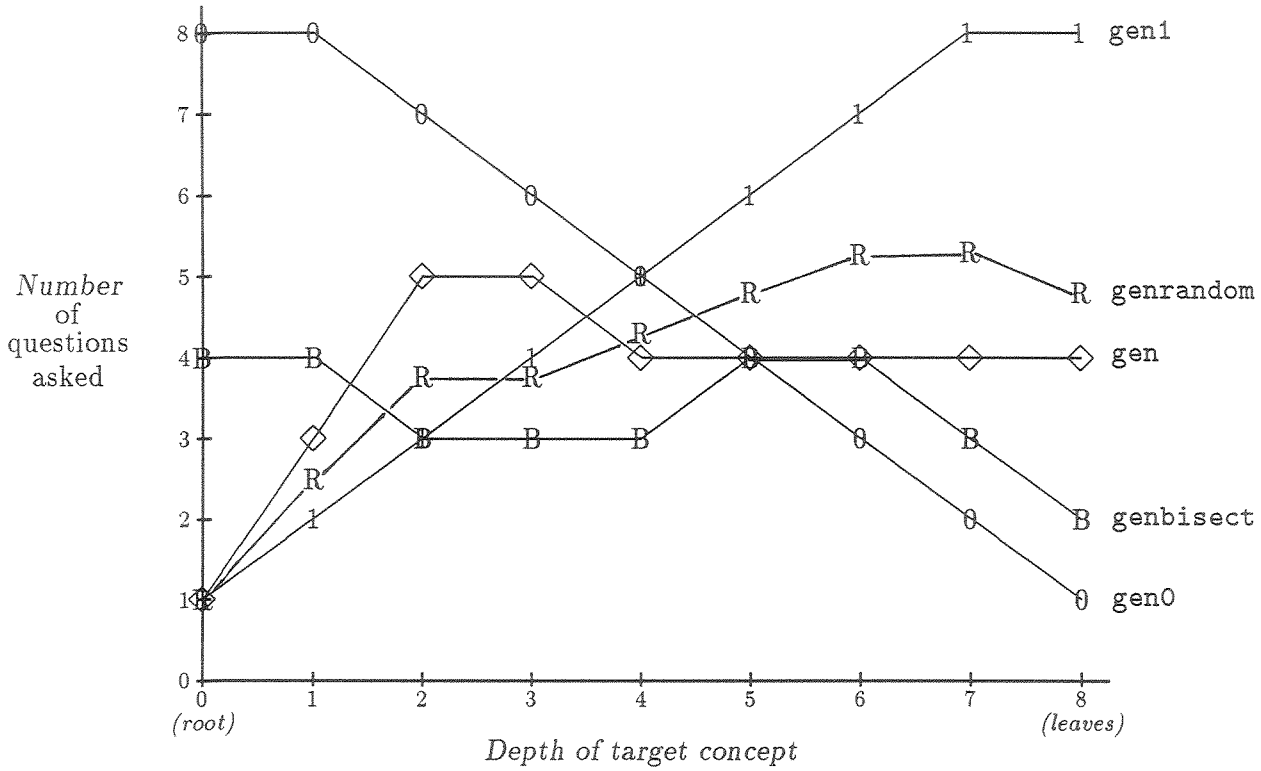

Figure 3: Results for 9-level 511-node tree.

oracle as positive or negative, is guaranteed to cause an update to G and/or S, thus shrinking the version space.

As Figure 3 shows, `genrandom` performs best for target concepts near the root. Because of its random nature it tends, in successive questions, to select leaves from different subtrees of G. The closer that the target concept is to the root, the higher the probability of getting a positive response from the oracle. Each positive response can generalize S several levels and thus very quickly collapse the version space to a single node. In the opposite situation where the target concept is near the leaves, `genrandom` does not perform as well. Here, the best questions to ask would be about nodes closely related to known positive instances. However, `genrandom` selects randomly from the space of leaves and therefore asks many questions.

4.2 gen1

The guiding principle behind this generator is to generate questions that have the potential for making the greatest changes in the version space. The generator begins by building a list of the immediate children of G. A node is chosen from this list that has no descendants in S. Finally, a leaf descendant of this node is selected as the question to pose to the oracle. So, in this situation the hypothesis node is a child of G. A positive response from the oracle can cause S to be generalized up through many levels. A negative response causes G to be specialized only one level, but in so doing it eliminates one or more large subtrees of the

version space.

The performance of gen1 shows a characteristic ascending slope. It asks the fewest questions for target concepts at the root and the most questions for concepts at the leaves. In fact, of all the generators, it is the worst performer for leaf concepts. The reason is that each question will receive a negative response from the oracle and cause G to be specialized only one level. This is an example of the worst "intelligent" question — a question that narrows the distance between G and S by only one level. This clarifies that what is important in any experiment generator is to reduce the number of levels between G and S. While it may seem appealing to eliminate a large subtree of the version space (by specializing G by one level), it is better to ask a question that has the potential to cause G and S to converge by more than one level.

4.3 gen0

This generator is the dual of gen1 in that it has exactly the opposite bias. It selects a leaf node that is a close relative (a sibling or, if necessary, a cousin) of a previous positive training instance. Effectively, the hypothesis node is a parent of an element of S (and a leaf descended from this hypothesis node is selected). A negative response from the oracle can cause G to be specialized many levels; a positive response causes S to be generalized only one level.

As Figure 3 shows, gen0 has a characteristic descending slope, the opposite of gen1. gen0 performs the best for leaf target concepts since the first question causes G to be specialized many levels. For target concepts near the root, gen0 generates the worst "intelligent" questions and therefore S gets generalized only one level at a time. As we shall see later in the performance comparisons, gen0 is one of the best overall generators, in spite of its poor performance for concepts approaching the root. The reason is simple — there are many more nodes near the leaf level (where it performs well) than the root level.

4.4 gen

This generator is an attempt to combine the strengths of gen0 and gen1 without their weaknesses. Its strategy is as follows:

- If the previous training instance was positive, that supports the possibility that the target concept is near the root, so use the gen1 strategy.
- If the previous training instance was negative, that supports the possibility that the target concept is near the leaves, so use the gen0 strategy.

This generator is unique among all the other generators in that it uses the *history* of the previous training instance.

As Figure 3 shows, its performance is quite good. It avoids the bad worst-case behaviors of gen0 and gen1 and its performance is much more consistent over the range of depth of target concept. Its performance for leaf target concepts is not as good as gen0 for a simple reason. Part of gen's behavior is due to the fact that the version space algorithm is always "primed" with a "first positive training instance". Thus, gen will *always* begin with the gen1

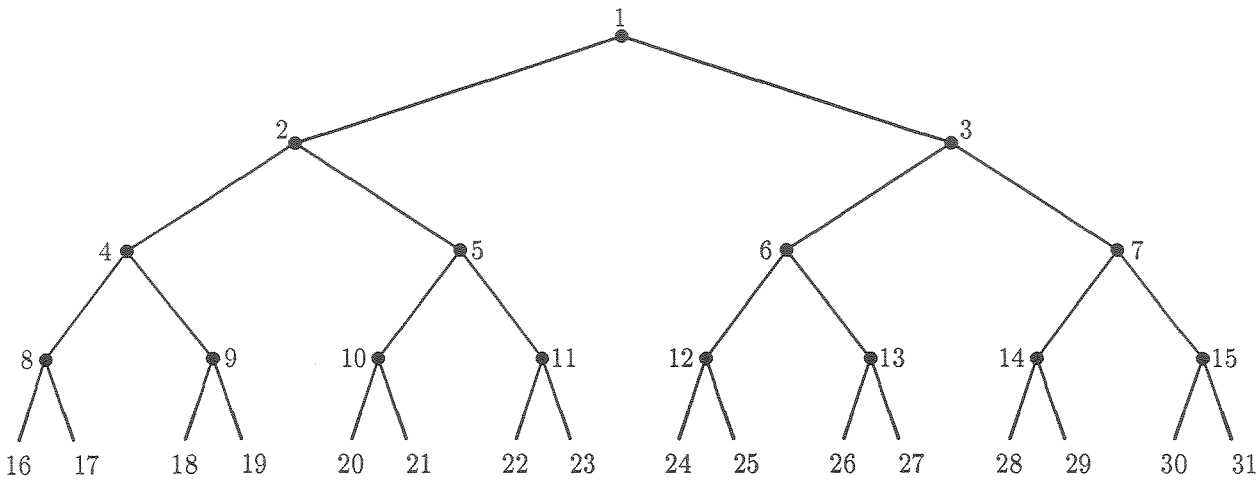

Figure 4: Sample concept hierarchy tree.

strategy, which is the correct one to use for target concepts near the root. This explains why it performs better near the root than near the leaves. In general, `gen` will alternate between the `gen0` and `gen1` strategies as it homes in on the target concept.

4.5 genbisect

As described above, `gen` repeatedly chooses between two very biased strategies. Intuitively, however, it seems that an experiment generator should *not* be biased if it is to perform well overall. `Genbisect` is such a generator. Its strategy, which is roughly analogous to binary search, was inspired by Mitchell's use of the term "bisect". The algorithm is easily explained by example.

Consider the binary tree shown in Figure 4. Assume that the target concept to be learned is "11". The following steps illustrate the alternation between the experiment generator (EG) and the version space algorithm (VSA) as the concept is learned.

- EG produces a *first positive instance* of "22".
- VSA initializes the version space with $S = \{ 22 \}$ and $G = \{ 1 \}$.
- EG constructs a path from S to G: { 22, 11, 5, 2, 1 }. It takes the middle element (5) and finds a leaf descended from it that is not also descended from any lower element on the path (i.e., not descended from 22 or 11). In this case it generates "20" as a new training instance.
- The oracle classifies "20" as a negative instance. VSA updates $G = \{ 11 \}$ (S remains the same).
- EG constructs a path from S to G: { 22, 11 }. It takes the "middle" element (11) and finds a leaf descended from it that is not also descended from any lower element on

the path (i.e., not descended from 22). In this case it generates "23" as a new training instance.

- The oracle classifies "23" as a positive instance. VSA updates $S = \{ 11 \}$ (G remains the same). Since S and G now match, the concept has been learned.

The target concept, which could have been any one of the 31 nodes, has been learned with 3 training instances (a first positive training instance plus two questions). In the worst case, an intelligent experiment generator would learn every concept in at most 5 questions (*i.e.*, one question per level of the concept hierarchy tree).

The performance results confirm the value of the bisection strategy. Although `genbisect` is not quite as good as `gen1` at the root level and not quite as good as `gen0` at the leaf level, it is well-behaved at all levels. Appendix A offers a proof that, as an iterative bisection strategy, `genbisect` is optimal when it chooses the node midway on the path between G and S.

5 Performance and Bias

There are two ways of judging overall performance. The first way assumes that the target concept can be at any *level* with equal probability. In this case it is appropriate to take the average of the average number of questions asked at each level. By that metric, the generators are ranked as shown in Table 1.

Generator	*Average*
`genbisect`	3.33
`gen`	3.77
`genrandom`	3.94
`gen0`	4.88
`gen1`	4.88

Table 1: Average number of questions per level

The other way to judge overall performance assumes that the target concept can be at any *node* with equal probability. In this case we must take the average number of questions over all nodes. This latter measure favors generators that do well near the leaves since there are many more nodes at or near the leaves than at or near the root. A binary tree, for instance, has more leaves than all other nodes combined. By this metric the generators are ranked as shown in Table 2. So, even though `gen0` has a bad worst-case behavior near the root, it outperforms all others because it is biased in favor of concepts near the leaves.

This observation about the importance of good performance at the leaf nodes stimulated a modification in our initial version of `genbisect`. Whenever the path from S to G contains an odd number of nodes, `genbisect` is able to select the exact middle of the path. However, when there is an even number of nodes in the path, there are actually two candidate nodes, neither of which is the exact middle. Our initial algorithm arbitrarily selected the "middle

Generator	*Average*
`gen0`	1.98
`genbisect`	2.69
`gen`	4.01
`genrandom`	4.92
`gen1`	7.51

Table 2: Average number of questions over all nodes

node" closer to G. When we changed it to select the node closer to S, the average performance over all nodes improved significantly because it asked one less question on leaf concepts. (It also asked one more question at the root, but that's a good tradeoff since there is only one root node.) It is this latter version of `genbisect` whose performance is reported in this paper.

6 Directed Acyclic Graphs

All of the results reported so far deal with a concept hierarchy *tree*. In general, the version space algorithm allows the concept hierarchy to be a directed acyclic graph (DAG). To test the effect of a DAG versus a tree, we modified the 511-node tree so that every third node gained an extra arc linking it to a node that was previously a nephew. Figure 5 shows the first 5 levels of the DAG with the extra arcs as dashed lines.

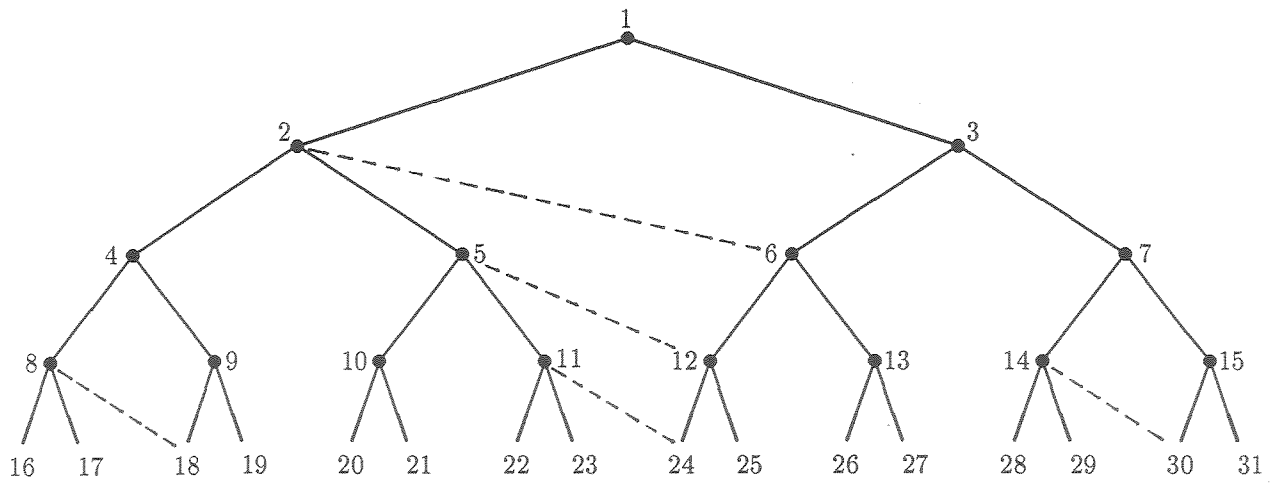

Figure 5: Sample concept hierarchy DAG.

In addition to testing the performance of the existing experiment generators, we devised an additional generator (described below) to test a new idea.

6.1 geninterior

Applications of the version space algorithm have generally assumed that only leaves can be used as training instances. However, if the learner was allowed to ask questions about interior (non-leaf) nodes, then it could ask directly about a hypothesis node rather than asking indirectly through a leaf of the hypothesis node. This is what `geninterior` does; it is different from `genbisect` only in that it asks questions about the hypothesis node directly rather than indirectly. This provides no advantage in a tree since there is always a single path between a hypothesis node and a leaf descended from it. However, this is not true in a DAG. As we shall see, `geninterior` outperforms `genbisect` in a DAG.

In all previous examples the questions posed to the oracle were of the form *"Is this node an instance of the concept?"* and the oracle always classified the training instance as "+" or "−". Now, however, it is possible that `geninterior` will ask about a node (the hypothesis node) that is more general than the target concept. Since this node has some descendants that *are* instances of the target concept and other descendants that are *not*, the oracle cannot respond with either "+" or "−". For this case we have modified the oracle to respond with "?".

The version space algorithm, of course, only knows how to update G and S for positive and negative training instances. Can the version space be updated on an indeterminate training instance? The answer is "yes", but not in the ordinary way. Since an indeterminate training instance is more specific than G (and more general than the target concept) then G can be specialized. We developed the following new update rule to do this:

> **Update Rule:** *Let G = { children of the training instance }. Eliminate from G any element that is more specific than any other element of G (subsumption test). Retain in G only those elements that cover all members of S.*

6.2 DAG Performance Results

Since our 511-node DAG is not symmetric like the 511-node tree, the number of questions asked can vary from node to node within a level. Accordingly, we ran each generator on the entire DAG (each node was designated, in turn, as the target concept). The performance results are shown in Figure 6 and tabulated in Tables 3 and 4. As before, Table 3 ranks the generators according to the average number of questions asked per level and Table 4 ranks them according to the average number of questions over all nodes. We discuss only the latter ranking here since it seems the fairer basis for comparison.

As Table 4 shows, the two bisection-based generators (`genbisect` and `geninterior`) clearly outperform the next closest competitor. Also, `geninterior` performs slightly better than `genbisect` for concepts near the root. With its ability to ask directly about hypothesis nodes, `geninterior` avoids the problem facing all other generators that use leaf training instances in a DAG — the problem that any given leaf training instance may imply more than one hypothesis node. The problem diminishes as the target concept gets closer to the leaves since there are less likely to be multiple paths from leaf to target.

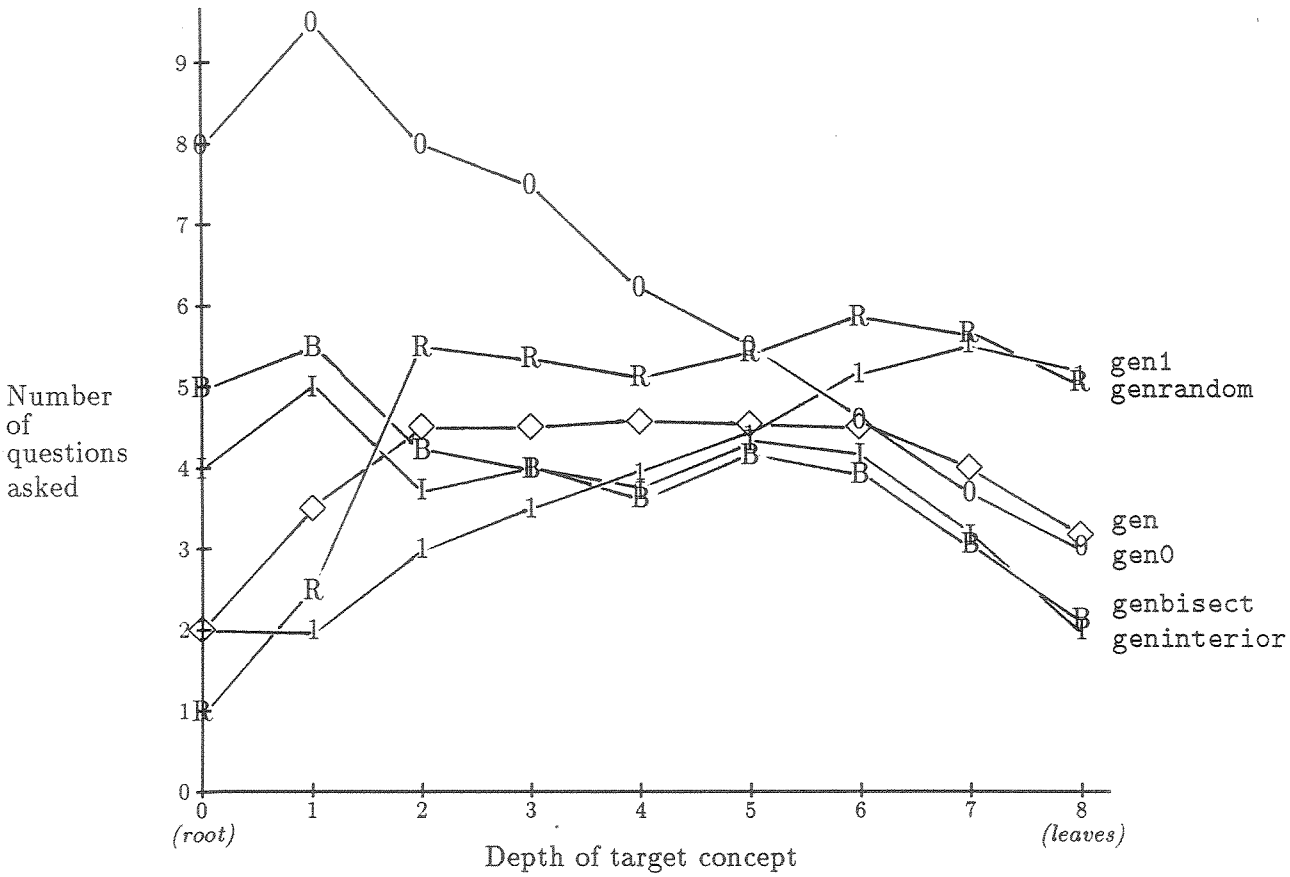

Figure 6: Results for 9-level 511-node DAG.

Interestingly, one of the generators (`gen1`) performed significantly better in the DAG than in the tree (5.08 questions versus 7.51). Recall that `gen1`'s strategy causes it to specialize G by one level with each negative training instance in a tree; the number of questions that it asks is proportional to the depth of the target concept. So why does it perform better in a DAG? The reason is that the extra parent-child links in the DAG can cause G to be specialized by more than one level per question. Consider an example with the DAG in Figure 5 where the target concept is node 8. After the first positive training instance $G = [1]$ and $S = [16]$. `Gen1`'s first question would be node 24, which the oracle would classify as negative. In the *tree* G would have been specialized to [2] but in the *DAG* it must be specialized to [4] since node 2 is (now) an ancestor of node 24. Thus, G has been specialized by more than one level. Nonetheless, `gen1` is still a poor performer overall because of its root-bias.

Generator	*Average*
`geninterior`	3.79
`gen1`	3.85
`gen`	3.92
`genbisect`	3.96
`genrandom`	4.62
`gen0`	6.23

Table 3: Average number of questions per level (in the DAG)

Generator	*Average*
`geninterior`	2.82
`genbisect`	2.84
`gen`	3.71
`gen0`	3.80
`gen1`	5.08
`genrandom`	5.33

Table 4: Average number of questions over all nodes (in the DAG)

7 Multiple-Attribute Concepts

All of the preceding discussion has dealt with single-attribute concepts; how do we handle multiple-attribute concepts? Figure 7 shows a 2-attribute example. Since the different at-

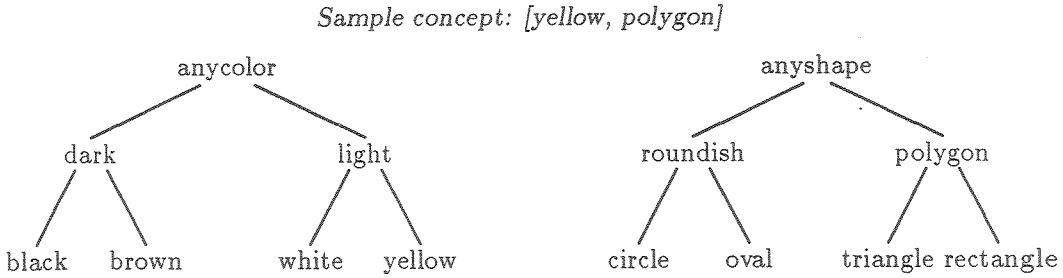

Figure 7: Concept trees for 2-attribute concept

tributes do not interact (a key assumption in this domain), the best strategy is "divide and conquer". Specifically, this means that given a first positive training instance such as [*yellow, triangle*], the experiment generator will modify only one attribute at a time. In fact, the program is written so that G and S must be converged on the first attribute before moving on to the second attribute. This strategy avoids the problem of credit/blame assignment. So, the performance of any generator on a multi-attribute concept is simply the sum of the number of questions asked on each of the separate concept hierarchies.

8 Conclusions

For learning in the version space, the results clearly show that it is possible to do better than random questioning. The technique described herein for bisecting the version space (*i.e.*, selecting a hypothesis node equidistant between the G and S boundaries), yields an experiment generator for concept learning that asks far fewer questions. As embodied in the `genbisect` and `geninterior` experiment generators, the technique is domain-independent, computationally simple and unbiased[2]. However, given some *a priori* domain knowledge (such as the fact that the target concept is near the root), a biased generator such as `gen1` can outperform `genbisect`. We believe that the bisection technique is optimal given no *a priori* knowledge.

The ability to pose questions about non-leaf nodes requires: (1) an oracle that can say "I can't classify this concept as + or −" and (2) a new update procedure that specializes G in this case. The resulting experiment generator (`geninterior`) performs better on irregular concept hierarchy graphs than any generator that poses questions about leaf nodes only.

The idea of using the results of a previous training instance (history) is very useful in selecting between biased strategies, as in `gen`. However, for an unbiased strategy such as bisection, history plays no role. After all, the results of every previous training instance are summarized in G and S by the version space algorithm.

The experiment generation strategies described in this paper are limited to classical concept definitions in which concepts are ordered by the *more-specific-than* relation and in which the attributes of a multiple-attribute concept are independent of one another. Unfortunately, this severely limits the range of applicability because many real-world concepts do not admit such classical definitions.

9 Acknowledgements

We wish to thank Professor Bruce Porter for suggesting this particular problem in "learning from experimentation" and for his thoughts on using the results of previous training instances. We also commend Philippe Alcouffe and Nicolas Graner on a fine Prolog implementation of the version space algorithm; it made our job much easier.

[2] Although it *sounds* appealing to say that a generator is unbiased, this is not necessarily desirable. Russell and Grosof [7] point out that it is better to have an explicit semantics for bias. They show how the process of learning a concept from examples can be implemented as a first-order deduction from the bias and the facts describing the instances. This has the following advantages: 1) multiple sources and forms of knowledge can be incorporated into the learning process; 2) the learning system can be more fully integrated with the rest of the beliefs and reasoning of a complete intelligent agent.

10 References

1. Mitchell, Utgoff and Banerji, 1983. "Learning by Experimentation" in *Machine Learning, Vol. I*, editors R. S. Michalski, J. G. Carbonell and T. M. Mitchell.

2. T. G. Dietterich, 1984. *Constraint Propagation Techniques for Theory-Driven Data Interpretation*, HPP-84-46, Stanford University.

3. Sammut and Banerji, 1986. "Learning Concepts by Asking Questions" in *Machine Learning, Vol. II*, editors R. S. Michalski, J. G. Carbonell and T. M. Mitchell.

4. Cohen and Feigenbaum, 1982. "Meta-DENDRAL" in *AI Handbook, Vol. 3*, pages 428–437.

5. T. M. Mitchell, 1982. "Generalization as Search", *Artificial Intelligence*, Vol. 18, No. 2, pages 203–226, March 1982.

6. Philippe Alcouffe and Nicolas Graner, 1985. *An Implementation of the "Candidate Elimination" Algorithm.* CS W395 report, available through Prof. Bruce Porter, Computer Sciences Department, The University of Texas at Austin.

7. Stuart J. Russell and Benjamin N. Grosof, 1987. *A Declarative Approach to Bias in Concept Learning.* Proceedings, Sixth National Conference on Artificial Intelligence (AAAI-87), pages 505–510.

A Proof of Optimality

The following proof shows that for an incremental bisection algorithm such as `genbisect`, the best node to pick as the hypothesis node on the path from S to G is indeed the node halfway in between. This proof applies only to the case where the target concept may be at any *level* with equal probability.

1. For concept hierarchy trees, when we are given the initial leaf node (the first positive training instance), we are assured that the target concept is on the unique path from the root node to the given leaf node.

2. We now abstract the problem to:

   Given:
   a path (determined by endpoints S, G)
   a concept k ($S \leq k \leq G$);
   an oracle (which *knows* k);

   Find:
   k, where the only questions that the oracle can ask are: for point p, "is $p \leq k$?" or "is $p > k$?"

3. Hence, to determine k the following program can be implemented. It is assumed that $0 < c < 1$.

   ```
   do
      if  k ≥ S + (G − S)c then  S ← S + (G − S)c
      if  k ≤ S + (G − S)c then  G ← S + (G − S)c
   until (G − S) < 1
   print ("k =", S)
   ```

4. We observe that since our level-of-generality metric[3] is isomorphic to the natural numbers, the assignment should actually be $S := \lceil S + (G - S)c \rceil$ and $G := \lfloor S + (G - S)c \rfloor$. We shall, however, base the subsequent proof on the level-of-generality metric being isomorphic to the real numbers so as to make the average case analysis easier. Hence, the termination condition becomes $(G - S) < 1$.

5. **Average Case Analysis**
   The probability distribution is assumed to be uniform. Hence, for a path of length l, the probability of a point being chosen is $1/l$. Let the number of statements required to determine a concept be m ($m = m_1 + m_2$) where:

   m_1 = number of times the first assignment was executed;
   m_2 = number of times that the second assignment was executed.

   At any step, since the probability distribution is uniform, the probability that the statement is chosen is c; that the second is chosen is $(1 - c)$. Note that the point $S + (G - S)c$ first divides the path between S and G into two paths of ratio $c/(1 - c)$. Thus in the average case $m_1 = mc$ and $m_2 = m(1 - c)$.

[3] The *level-of-generality* metric refers to the level of a node, where the root is at level 0.

6. **Theorem:** `genbisect` is optimal when $c = 1/2$.
   After m steps the length of the path is $(G-S)c^{m_1}(1-c)^{m_2}$. The termination condition ensures that this length is at most 1.

   - $(G-S)c^{m_1}(1-c)^{m_2} \leq 1$
   - $\log(G-S) + m_1 \log c + m_2 \log(1-c) \leq 0$
   - $mc\log c + m(1-c)\log(1-c) \leq -\log(G-S)$
   - $m \leq -\log(G-S)/c\log c + (1-c)\log(1-c)$

   To minimize m we require:

   - $x = c\log c + (1-c)\log(1-c)$ be maximized;
   - $dx/dc = 1 + c - 1 - (1-c) = 0$
   - $c = 1 - c$

   Therefore, $c = 1/2$.

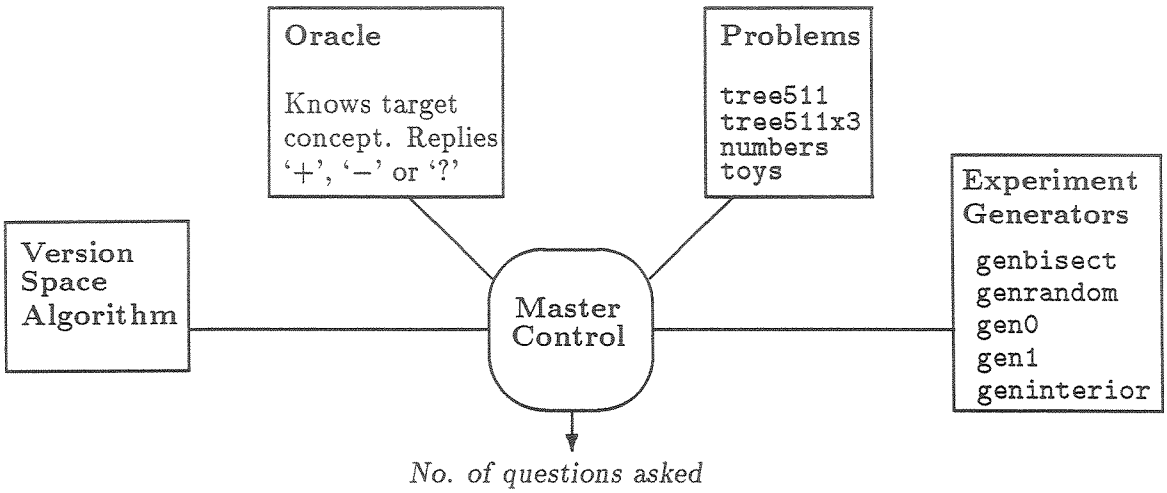

Figure 8: Testbed for experiment generators.

B Software

All of the software used in this project (which is shown in Figure 8) is available upon request. The software is written in Prolog; brief descriptions of the different files are provided below.

vsa A modified version of the Alcouffe and Graner implementation of the version space algorithm in which some clauses have been removed since they are modified and replaced by other files.

upd This contains the new rule for the version space algorithm to update G and S in the case of an indeterminate ("?") training instance.

oracle The oracle. It is the only module that knows the target concept. It classifies every training instance as "+", "-", or "?".

master This is the master control program that reads in the appropriate problem file and experiment generator and then controls the learning cycle involving the generator, oracle, and version space algorithm.

tree511 Contains a simple binary tree of 511 nodes.

tree511x3 The same as tree511 except that every third node has an extra arc linking it to a nephew. This provides a directed acyclic graph.

genask For testing/debugging/tutorial purposes, this is a generator that asks at the terminal for a training instance (instead of generating one by itself according to some algorithm).

genrandom, gen0, gen1, gen, genbisect, geninterior Each of these experiment generators is described in the text.