scribed here in place of the successive spaces implied by the definition above. It also minimizes the space requirement of a canonical string. It does require some language features, or subroutines, to extract the count as an integer, to determine its size. It also means that character comparison is harder to implement; equality of a character with one found in a string may mean either that the hoped for character has been found or it may mean that a relative tab count happens to have the same bit pattern as the desired character; reference to the previous character in the string is required to distinguish the two cases.

5. A print position consists of some non-zero number of character positions, occupying different half line vertical positions in the same horizontal carriage position. All but the last character position of a print position are followed by a backspace character and some number of HLF characters.

print position : = character position
   [BS [HLF]...character position]...

6. A character position consists of a sequence of graphic formers separated by backspace characters. The graphic formers are ordered according to the USASCII coded numeric value of the graphics they contain. (The first graphic former contains the graphic with the smallest code, etc.) Two graphic formers containing the same graphic will never appear in the same character position.

character position : = graphic former
   [BS graphic former]...

Note that all possible uses of a backspace character in a raw input stream have been covered by statements about horizontal carriage movements and overstruck graphics.

7. A graphic former is a possibly zero-length setup sequence of graphic controls followed by one of the 94 USASCII non-blank graphic characters.

graphic former : = [setup sequence] {one of the
94 USASCII
graphic characters}

setup sequence : = \{'RRS [BEL]
BR[ ]\}

8. A graphic setup sequence is a color shift or a bell (BEL) or a color shift followed by a bell. The color shift only appears when the following graphic is to be a different color from the preceding one in the message.

By virtue of the above definitions, the control characters HT, VT, and CR will never appear in a canonical stream.
A study of heuristic learning methods for optimization tasks requiring a sequence of decisions*

by L. ROWELL HUESMANN

Yale University
New Haven, Connecticut

INTRODUCTION

Learning is a broad term covering many different phenomena. It is convenient to segment learning into three different problems in induction: the collection and use of stochastic information on past performance in order to improve performance, the determination of which variables are relevant to the decisions being made, and the derivation of performance rules in the predicate calculus from the collected data. This study concentrates on the first problem.

THE ISSUES FOR INVESTIGATION

(a) Can a digital computer program significantly improve its performance on an optimization task of real-world complexity (and generalize that improvement to other problems of the same type) solely through ordinal feedback from intercomparisons of the solutions it has produced?

Most of the previous work in machine learning dealt with pattern recognition or game playing tasks. Yet these tasks have specific characteristics that differentiate their requirements for a learning mechanism from other tasks' requirements. Both are essentially win-loss or right-wrong tasks. In addition, in pattern recognition, feedback about the success of a decision is usually immediate. Yet many tasks have other than binary outcomes—that is, they are optimization tasks or problems in finding the "best" solution, according to some objective criterion, from a set of feasible solutions. Usually, the problem solver does not even know how well he can do. Consumer decisions, social decisions, and business decisions are often problems of this type.

With many optimization tasks one can obtain interval information about the relative worth of two solutions, however for others only an ordinal scale of solutions can be found. More important, it is often an order of magnitude easier for a program to decide whether one solution is better than another than for it to decide how much better. Hence, it is desirable to find a mechanism that can improve a program's performance solely from ordinal feedback.

(b) Can significant improvement occur if the task environment is characterized for the program by a vector of relevant stimulus variables (a state vector)?

Another characteristic of much of the previous work in machine learning is that most learning mechanisms have combined the stimulus variables in linear polynomials and selected a response on the basis of the values of the various polynomials' values. Many of these schemes are called stimulus voting procedures because each stimulus votes separately for a response.

The limitations of such linear machines are well known and have been analyzed in detail.1,2 What is particularly disappointing is the simplicity of some patterns that cannot be handled by linear machines. For example, consider the association pattern in Table 1. When the values of the two features are the same, response R1 is required; otherwise, R2 is required. Let us now show that linear discriminant functions cannot be used to make this classification.

Theorem: Linear discriminant functions do not exist for some very simple classifications of features. In particular none exist for the classification shown in Table 1. Proof: The theorem will be proved by assuming the linear discriminant functions do exist and finding a

* This paper is based on a Ph.D. thesis completed at Carnegie-Mellon University, Pittsburgh, Pa. The project was supported in part by United States Public Health Service grants MH-07722 and MH-30,068-01A1. The author is indebted to Mr. Herbert A. Simon for his advice and assistance.
contradiction. Let \( E_1 \) be the linear discriminant function for \( R_1 \):

\[
E_1 = C_n F_1 + C_m F_2
\]

Let \( E_2 \) be the linear discriminant function for \( R_2 \):

\[
E_2 = C_n F_1 + C_m F_2
\]

If linear discriminant functions exist that can make this discrimination, then

For \((F_1 = 1, F_2 = 2)\) and \((F_1 = 2, F_2 = 1)\)

\[
E_2 - E_1 = C_n F_1 + C_m F_2 - C_n F_1 - C_m F_2 > 0 \quad (1.1)
\]

For \((F_1 = 1, F_2 = 1)\) and \((F_1 = 2, F_2 = 2)\)

\[
E_1 - E_2 = C_n F_1 + C_m F_2 - C_n F_1 - C_m F_2 > 0 \quad (1.2)
\]

Substituting the values of the features gives from (1.1)

\[
2C_m + 2C_n - C_{11} - 2C_{12} > 0 \quad (1.3)
\]

and

\[
2C_m + 2C_n - C_{11} - C_{12} > 0 \quad (1.4)
\]

and from (1.2)

\[
C_n + C_m - C_{11} - C_{12} > 0 \quad (1.5)
\]

\[
-(C_n + C_m - C_{11} - C_{12}) < 0
\]

\[
C_n + C_{11} - C_{12} < 0 \quad (1.6)
\]

But adding (1.3) and (1.4) gives

\[
3C_n + 3C_m - 3C_{11} - 3C_{12} > 0
\]

\[
C_n + C_{11} - C_{12} > 0 \quad (1.7)
\]

Equation (1.7) contradicts (1.6). Since the conclusion of a correct line of reasoning has been a contradiction, the assumption that a linear discriminant function exists must be false, and the theorem is proved.

A mechanism that associated states of the environment with strategies or responses could learn such discriminations. Unfortunately, a state vector description requires a great deal more computer storage. For example, for \( R \) stimulus variables and \( N \) values per variable a parsimonious representation of the stimulus state requires on the order of \( N^2 \) storage cells. On the other hand, one needs only \( N^*R \) cells to represent the status of each stimulus variable independently of the other variables and only \( R \) cells to represent the stimulus situation as the value of a linear polynomial. However, psychological evidence indicates that humans seldom attend to more than a few environmental features at a time\(^3\) so a state-vector of low dimensionality might be a reasonable representation for a learning program. This is the representation we adopted.

The learning problem

We view the learning problem as one of associating states of the environment, defined by some set of stimulus variables to which the problem solver is attending, with strategies for performance. The strength of such associations can be represented by the entries in a table of connections or matrix whose rows represent stimulus states and whose columns represent strategies. We want to see if significant learning can be accomplished on a very complex optimization task if the stimulus environment is represented by a state vector of some of the most obvious relevant stimulus variables and only ordinal feedback is used.

THE TASK TO BE LEARNED

To avoid spending a majority of the programming effort on a performance program for solving a very general class of optimization tasks, it was decided to restrict the study to one specific task, the project scheduling task. A sample problem for this task is shown in Figure 1. The objective is to complete all the jobs in as short a time as possible by executing them in parallel. It is a difficult real-world task faced by management scientists, but it can be shown to be very similar to other optimization tasks requiring a sequential set of decisions, e.g., finding the minimum number of moves to checkmate, or the Traveling Salesman Problem. A task very similar to the project scheduling task was used by Fisher and Thompson\(^4\) in a study that suggested the learning technique we have used. One can view optimization tasks that require a sequence of decisions as problems in finding the shortest or longest path through a decision tree. A feasible solution is any path from the root of the tree to a terminal or goal node. The branches descending from a node represent possible decisions and the nodes represent the status of the “system” after a decision is made.
THE LEARNING TECHNIQUE

Given a state-vector representation of the task environment and a set of performance strategies, the learning mechanism must create a good (and generalizable) table of connections between stimulus states and strategies. An informal “hill climbing” procedure will be used to construct the table. Viewing learning as constructing a table of connections is not a new idea. However, unlike almost all previous learning programs, this one will have no way to make an absolute judgment about the utility of a solution. Since the problems to be attacked are optimization problems themselves, the learning program cannot determine when it has achieved the best solution. How will feedback be obtained?

The best previous solution will be designated as a benchmark solution and new solutions will be compared to it. If the new solution is better, the comparison is positive; if it is worse, the comparison is negative. A fairly sophisticated comparison procedure was developed to make comparisons feasible as frequently as possible during construction of a solution. Hence, one comparison corresponds to what is normally called one trial in the learning literature and one trial on a problem includes a whole series of comparisons. One can show that this technique can be applied (with a few restrictions) to almost any optimization task requiring sequential decisions.

THE DESIGN OF THE PROGRAM

This program, like most learning programs, should be viewed as two closely interacting routines—a performance routine and a learning routine. The routines were written in IPL-5. Let us first discuss the performance routine.

The performance routine

This routine is designed to find the shortest path through the tree of feasible solutions, i.e., feasible job schedules. Each level in the tree corresponds to a different time; each node in the tree specifies what jobs are completed, what jobs are currently being executed, and what jobs remain to be scheduled; each branch indicates the scheduling of a particular set of jobs. Hence, two geometrically different nodes may have the same meaning with different histories. Every path eventually leads to a node specifying that all jobs are complete. The objective of the performance program is to find a path through this tree that ends at the highest level terminal node (minimum time path).

The performance program uses a “depth first” approach to search. It looks ahead along a path through the tree until it detects a node where the path can be evaluated. Of course, there will be no evaluations during the production of the initial solution since there is no solution for comparison. At each node encountered in the look ahead process, the program must decide what branch to follow next. This is equivalent to choosing the jobs that should be scheduled at that time. When a node is reached that can be evaluated, the learning program is called in to compare the current path with the benchmark solution. The comparison is either indeterminate, positive—the new path is more desirable, or negative—the present solution is more desirable. If the comparison is indeterminate or positive, the performance routine looks ahead deeper along the current path. In addition, when the evaluation is positive, the current path is merged with the benchmark solution to form a new benchmark solution. On the
other hand, if the evaluation is negative, the performance routine abandons search along the current path. It may either return to the top of the solution tree and investigate a new path or look ahead deeper from the corresponding but preferred node on the present benchmark solution's path. By "corresponding node" we mean the node on the solution path that was used in the evaluation.

The tables of connections

In carrying out this procedure the performance program has to make two types of decisions. As mentioned above, following a negative comparison, the program must decide whether to back up to the start or continue from a point on the present benchmark solution; the program must also decide what branch to follow (what jobs to schedule) at each node encountered in the look ahead process. While the former of these decisions requires a general-problem-solving strategy, the latter decision requires a task-specific strategy. To select these strategies, the performance routine employs two tables of connections. One table links a state vector composed of characteristics of the current search situation to general strategies, in this case strategies specifying what to do after a negative comparison. The other table connects a state vector of task relevant variables to a set of task specific strategies, in this case job selection strategies.

Both of these tables are represented by tree structures in the computer's memory. A numerical value associated with Strategy $i$ at State-node $j$ will provide a measure of the past success of that strategy in State $j$ relative to the success of other strategies in that state.

Selecting a strategy

The information in a state node could be used in any of several ways to select a strategy. For example, if one wishes to select a good strategy, one might choose the strategy whose success value is the greatest of all the values at the node, or one might select a strategy probabilistically in proportion to the success values. On the basis of several pilot runs it was decided that the performance program should construct the initial solution on each run by selecting the strategies with the highest success values (ties are broken randomly). During the rest of the run the performance routine would select strategies probabilistically. Specifically, the probability of choosing Strategy $s_i$ whose success value in the current state is $v_i$, from $n$ strategies whose values in the current state are $v_1 \ldots v_n$ is given by

$$P(s_i) = \frac{v_i}{\sum_{k=1}^{n} v_k}$$

Built-in heuristics

The performance routine was not intended to begin as a completely naive problem solver. Certain general and task specific heuristics were built into the routine while other heuristics were introduced via the initial entries in the general-strategy table of connections. These heuristics were ones that most human problem solvers would have learned before ever attempting a problem of this type or ones that would be suggested by a cursory glance at the literature on the task. Foremost, among the general heuristics built into the program, is sub-goal evaluation. During the look-ahead process, the performance program asks the learning program to evaluate virtually every potential sub-goal—that is, every node on the current path that is on the same level as a node of the solution path. (In look-ahead searches on a typical problem twenty-eight sub-goals were evaluated for every evaluation of a complete path through the tree of feasible solutions.) A second built-in general heuristic is the program's "next event" approach to search. During the look-ahead process many nodes are encountered where no decisions need to be made. For example, no jobs can be scheduled at a node unless a job terminates there. Hence, the performance program jumps from node to node ignoring intervening nodes where no decisions need to be made. This heuristic speeds performance greatly, but it is dependent upon another heuristic—a task specific heuristic—included in the performance program. The performance routine always schedules as many jobs as resource constraints permit; so no new job can be scheduled until a job terminates and frees some resources. Such a heuristic is not without its drawbacks. There are a few situations where it prevents the program from searching a slightly superior branch. However, it is a heuristic with strong intuitive appeal, one that reduces the number of branches in the solution tree considerably, and one that permits implementation of the next event search process reducing the number of nodes to be analyzed during look-ahead.

Three heuristics dealing with search behavior were introduced through the initial values of the success terms in the general-strategy table of connections. These heuristics deal with what the program should do following the discovery that a branch presently being searched can only be inferior to the solution (negative comparison). The probability of "backing
up" to the start was made an inverse function of the depth that search had progressed into the solution tree and a direct function of the number of dead end branches encountered (negative comparisons and non-positive comparisons of complete paths). Thirdly, whichever "back up" strategy is selected, it should be tried several times consecutively before being abandoned.

Storing a solution

The performance program remembers only the present solution path and the path currently being searched. Both are stored as lists of scheduled jobs separated by time markers. The jobs preceding the nth time marker are the jobs being executed at time n. Associated with the list representing the path currently being searched is a list of the values in the tables of connections that have been selected during the search—that is, the values corresponding to the state-strategy pairings used to produce the path. Whenever a strategy is used, its value in the current state is added to this list. Hence, all the cells that the learning routine will modify are contained on this list.

Figure 2—A flow chart of the performance routine.

Figure 3—A flow chart showing the steps the learning routine takes when called upon to compare part of a new solution with the bench mark solution.

A gross flow chart of the performance procedure is presented in Figure 2.

The learning routine

The learning program evaluates paths through the tree of feasible solutions by comparing them with the bench mark solution (best solution so far), and it alters the tables of connections on the basis of these evaluations.

Comparing solutions

How does the program determine which of two paths is preferred? Path Z1 up to node x is preferred to path Z2 up to node y if node x and y are at the same level in the tree of feasible solutions and if node x "dominates" node y. A node, one should remember, specifies the set of jobs currently completed and the set of jobs currently being executed. To say node x on Z1 dominates node y on Z2 means that (a) all jobs
TABLE II—These Sample Schedules Illustrate the Construction of a New and Superior Bench Mark Solution (Z4) Out of the Old Bench Mark (Z2) and a New Partial Solution (Z1).

<table>
<thead>
<tr>
<th>TIME</th>
<th>Z1</th>
<th>Z2</th>
<th>Z3</th>
<th>Z4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>job 1 job 2</td>
<td>job 1 job 4</td>
<td>job 1 job 2</td>
<td>job 1 job 2</td>
</tr>
<tr>
<td>2</td>
<td>job 3 job 4</td>
<td>job 2</td>
<td>job 3 job 4</td>
<td>job 3 job 4</td>
</tr>
<tr>
<td>3</td>
<td>job 5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>job 6 job 7</td>
<td>job 5</td>
<td>job 6 job 7</td>
<td>job 8</td>
</tr>
<tr>
<td>6</td>
<td>job 8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LIST REPRESENTATION OF Z1:

Z1—0
job 1
job 2
time mark
job 3
time mark
job 4
time mark
job 5
time mark—0

scheduled on Z1 prior to x are completed by x, (b) all jobs on Z2 that are completed by y or being executed at y are completed by x on Z1 (from (a), any job on Z1 prior to x is completed by x), and (c) there is at least one job completed on Z1 prior to x that is not on Z2 prior to y. From this definition if any node x on Z1 dominates its corresponding node y on Z2 (the node at the same level), then combining Z1 prior to x with Z2 after y produces a new path at least as short as Z2 and in most cases shorter. Hence, Z1 prior to x is preferred to Z2 prior to y. One should be careful to clearly understand these statements as they are essential to the learning method. They form a task specific algorithm for judging partial schedules. To verify that the new path will indeed be no longer, one simply recognizes that Z2 after y can always be added onto Z1 before x without any changes since all jobs on Z1 are completed at x. Furthermore, at least one job on Z2 after y has already been executed and can be deleted. If this deletion (or deletions) shortens Z2 after y the new path will be shorter. Consider as an example the schedules Z1 and Z2 in Table 2. At time 4 the node on Z1 dominates the corresponding node on Z2. As a result Z1 and Z2 can be combined into the new schedule Z3. By deleting “job 5” which had already been completed on Z1 and moving the other jobs up in time, a shorter schedule Z4 was then produced.

The large majority of evaluations turn out to be indeterminate. For example, during training on a typical problem about 94% were indeterminate. When the comparison is negative (the present bench mark solution is preferred), the tables of connections are not altered and control is returned to the performance routine which decides whether to look ahead from the corresponding node of the bench mark or back up to the top of the tree. When the comparison is positive (the current path is preferred), the learning program alters the tables of connections and constructs a new bench mark solution.

Altering the memory structures

Altering the tables of connections is fairly trivial. Remember that during look ahead a list is maintained of all the success terms associated with the selected state strategy pairings. This requires very little storage, only one cell for each decision made since the last
positive or negative comparison. To positively reinforce
the state-strategy pairings participating in the con­
struction of a better solution, each element of this list
is simply incremented. On the basis of pilot studies we
selected an increment of 3 over smaller values. Larger
values might produce more rapid learning but also less
stable. Obviously, an entire study could be devoted to
finding the optimal value for this increment. With an
increment of 3 the probability of selecting each strategy
is altered as follows.

Let \( P_t(s_i/R) \) be the probability at time \( t \) of selecting
strategy \( s_i \) in state \( R \).

\( v_j \) be the success value associated with the
\( j \)th strategy in state \( R \) at time \( t \).

\( n \) be the total number of strategies. Then, as mentioned
earlier,

\[
P_t(s_i/R) = \frac{v_i}{\left( \sum_{j=1}^{n} v_j \right)}
\]

and if strategy \( s_i \) is reinforced in state \( R \) at time \( t \)

\[
P_{t+1}(s_i/R) = \frac{(v_i + 3)}{\left( \sum_{j=1}^{n} v_j + 3 \right)}
\]

\[
P_{t+1}(s_k/R) = \frac{v_k}{\left( \sum_{j=1}^{n} v_j + 3 \right)}, \quad k \neq i
\]
or letting

\[
M = \sum_{j=1}^{n} v_j
\]
at time \( t \)

\[
P_{t+1}(s_i/R) = \frac{M}{M + 3} P_t(s_i/R) + \frac{3}{M + 3}
\]

\[
P_{t+1}(s_k/R) = \frac{M}{M + 3} P_t(s_k/R)
\]

These changes will be called positive reinforcement.

Consistent decision making during learning

This completes the description of the learning routine.
One very important addition was made to the learning
scheme as a result of some early failures in the pilot
studies.

**Principle:** While exploring a path through the tree
of feasible solutions, a performance program used
with a learning routine should employ the same
strategy every time the same state occurs (make
the same decision in the same situation) until the
path has been successfully evaluated (positively or
negatively).

When this principle is not adhered to, credit assignment
becomes almost impossible. Conceivably, all the strat­
egies could be used in the same state before an evalu-
ation occurred. In this case the bad strategies may
mask the good strategies, and one has no way to
distinguish between them. Hence, it is not sufficient
to “select a strategy in proportion to its past successes.”
One must first check to see if a strategy has already
been paired with the current state, and, if so, use that
strategy.

SELECTING STRATEGIES AND FEATURES

One might well argue that the major portion of this
program’s work is done by the programmer when he
selects the stimulus features for attention and the
potential strategies for use. Yet this is exactly what
happens to the human beginner. He generally derives
his first ideas about strategies and features from a
teacher, a book, or his experience with other similar
tasks. The features and strategies that we selected for
use were simple ones that would occur to anyone who
made a cursory glance at the literature on scheduling
problems. Within the program the features and strat­
egies were represented as lists of components in such a
way that new strategies or features could be synthe-
sized. Later we will see how this learning mechanism
could employ its feedback to eliminate poor strategies
or features and introduce new ones.

Five task-specific strategies and three features of 3,
3, and 4 values were used initially. Hence, there were
3*4*3 or 36 state nodes in the task-specific table of
connections. Each state, of course, really represented
a broad class of stimulus situations. With five strategies
per node the total storage requirement of the task
specific table of connections was only 463 IPL–5 cells
or 926 32-bit words. All the success values in this table
were initially set at 10. Other smaller values were tried
during the pilot studies and found to change the per­
formance routine’s behavior too radically in early
training.

The general-strategy table of connections used in
these experiments was employed only to choose be­
tween two search strategies. The “previous-strategy”
feature thus had two values while the other two features
had three value classes. Hence, there were 18 state
nodes requiring 133 IPL–5 cells or 266 32-bit words.
This means that the two tables’ total storage re­
quirement was 1,192 computer words. The initial success
values in the general-strategy table were assigned to implement certain heuristics as discussed in the section on the performance routine.

EXPERIMENT I

To answer questions (a) and (b) we tested this program’s learning ability on three project scheduling problems.

The dependent variable on which a learning mechanism should be evaluated is improvement in performance not the quality of performance. We want to demonstrate that the proposed learning mechanism, using only ordinal feedback, can learn what strategy to apply in what state so that the performance program performs significantly better on the training problem and on other problems of the same type.

Fifteen project scheduling problems (unbiased in any obvious manner) were generated randomly by the computer to find three that satisfied hardware and complexity constraints (most were too simple).

We will call these Problems 1, 2, and 3. The program was trained on Problem 1, trained more on Problem 2, and tested for ten minutes on Problem 3. Then we retrained the program from scratch on Problem 3 and tested it on Problem 1. No negative reinforcement was applied in this experiment. If the benchmark solution was definitely superior, the new path was abandoned and the program selected a general strategy telling it what to do next.

Results

The training significantly improved the performance program after only a moderate number of positive
reinforcements. The improvement generalized to problems other than the training problem.

Figures 4.1 to 4.3 and 6.1 to 6.2 contain learning curves showing the improvement. The base solution to a problem is the average solution produced by random strategy selection. The improvement can be measured quantitatively by the ten-minute solution rate. The rates are shown in Table 3. A Mann-Whitney U test confirmed a highly significant difference ($p < .0001$) in rates on training and test trials.

Each segment of the learning curves in Figures 4.1 and 6.1 represents the performance from creation of an initial benchmark by using the highest valued state-strategy pairings until the program has not improved the benchmark in a specified time period. The benchmark is erased before a new segment starts. These segments are called training trials, but within any one of them there are many comparisons of solutions which may result in reinforcements. About 94% of all comparisons were indeterminate, i.e., neither the benchmark nor current solution was preferred. On the average 9 different state-strategy pairs were evaluated in a determinate comparison. One inevitable characteristic of an ordinal feedback system is that as learning progresses within a trial, positive comparisons become less frequent, and negative comparisons become more frequent. Altogether, during the three training trials on Problem 1, there were 449 positive reinforcements of state-strategy pairs, while, during the training on Problem 3, there were 142 positive reinforcements of state-strategy pairs. The change in one individual row in the table of connections is displayed in Figure 5.1. One can see that an equilibrium was reached early in the trial. The changes in the table of connections can also be measured in terms of the entropy of the table...
of connections. The entropy of the table trained on Problem 1 was reduced from 83.6 bits to 71.7 bits during training. This was 93% of the maximum possible reduction for the number of reinforcements. The final table of connections for training on Problem 1 is summarized in Table 4.

On the basis of relatively brief exposure to one optimization problem the performance program's table of connections was changed so that the program produced good solutions significantly more rapidly. This learning generalized to two other problems of the same type. Training from a na"ive state on these problems in turn improved performance on the original problem. Hence, significant learning is possible on a very complex optimization task with ordinal feedback and a state-vector representation of a reduced task environment.

EXPERIMENT II

(c) Does improvement occur more rapidly if the program changes its structure following its failures to improve its performance as well as after its success?

Having demonstrated that a learning mechanism based on ordinal feedback and using a state-vector representation will work, we can turn to the central issue in this study: should negative reinforcement (or error correction training) be used in a learning mechanism for optimization tasks?

The large majority of trainable pattern classifiers and game playing programs have used error correction training alone or in conjunction with positive reinforcement. This is somewhat surprising since the weight of evidence from psychology seems to indicate that positive reinforcement plays the most important role in learning while negative reinforcement may speed learning slightly by eliminating incorrect responses or may not help at all. Furthermore, we assert that error correction training is useful only if the learning program receives feedback data on an interval or ratio scale; feedback on an ordinal scale, as one receives in optimization tasks, while sufficient for positive reinforcement, is not sufficient for negative reinforcement (error correction training). In fact, error correction training or negative reinforcement should adversely affect the

| TABLE III—Solution Rates During Experiment I (Positive Reinforcement) |
|-------------------------|---------|---------|
|                         | Trial 1 | Trial 2 |
| Training on Problem 1:  |         |         |
| 1                       | 0.9     | 2.2     |
| 2                       | 1.2     | 2.2     |
| 3                       | 2.9     | 3.1     |
| Additional training on Problem 2: |         |         |
| 1                       | 2.4     | 2.9     |
| 2                       | 2.9     | 3.9     |
| Test on Problem 1:      |         |         |
| 1                       | 5.0     |         |
| Test on Problem 3:      |         |         |
| 1                       | 5.1     |         |

<table>
<thead>
<tr>
<th>Second training series (Run 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trial 1</td>
</tr>
<tr>
<td>Training on Problem 3:</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>Test on Problem 1:</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>Test on Problem 3:</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>FEATURE</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>BEGINNING</td>
</tr>
<tr>
<td>Depth in Middle</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>END</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>TIME NEEDS</td>
</tr>
<tr>
<td>EXECUTABLE JOBS</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>ABOVE</td>
</tr>
<tr>
<td>AVERAGE</td>
</tr>
<tr>
<td>VARIANCE</td>
</tr>
<tr>
<td>OF CRITICALITY</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>ABOUT</td>
</tr>
<tr>
<td>AVERAGE</td>
</tr>
<tr>
<td>ABOVE</td>
</tr>
<tr>
<td>AVERAGE</td>
</tr>
<tr>
<td>MEANS</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

**Strategies:**
- S21: Schedule job with minimum resource demands
- S22: Schedule job with maximum time demand
- S23: Schedule job with maximum criticality
- S24: Schedule job whose time demand is closest to the remaining time for a scheduled job
- S25: Schedule job with maximum resource demands

Before showing why error correction training should hamper this type of learning, we need to review three key features of our ordinal learning program.

(a) The program possesses a preference routine that enables it to compare parts of new solutions with a benchmark solution.
(b) The program implements positive (or negative) reinforcement by incrementing (or decrementing) those cells in the table of connections (by Cpos or Cneg) that contributed to the new solution.
(c) The program uses strategy $j$ in state $i$ with a probability equal to $v_{ij}/\sum_{y} v_{xy}$ where $v_{xy}$ is the value of the cell corresponding to state $x$ and strategy $y$. Hence, the summation is over all strategies in state $i$.

Now we can state the theorem leading to our conclusion that error correction training will fail for any sequential
optimization task representable as finding the optimal path through a tree of feasible solution.

Theorem: For any ordinal-feedback learning procedure possessing characteristics a, b, and c, error correction training will decrease the probability of selecting the “best” strategy or response in each stimulus situation as soon as

1. The “best” strategy is being used in over 50% of the situations encountered
2. The probability that the benchmark will be preferred to a new solution is greater than \( C_{pos} / (C_{pos} + C_{neg}) \).

In less formal terms, when the benchmark solution and the table of connections have both become pretty good, negative reinforcement will begin to make the table of connections worse. Though this theorem will be proved for a program with characteristics a, b, and c, the reader should realize that the theorem (in slightly different form) will hold for viable alternatives to characteristics b and c. The central problem is that ordinal feedback becomes unreliable as the benchmark improves.

Proof:

Let \( C_{pos} \) be the increment for positive reinforcement. \( C_{neg} \) be the decrement for negative reinforcement. 

\( P \) be the probability that the benchmark solution is preferred after a determinate comparison.

\( V(i, j) \) be the entry in the table of connections corresponding to the ith state and the jth strategy (at time t).

\( E_t(V) \) be the average value of \( V(i, j) \) over all state-strategy pairs used in constructing a new path.

\( q \) be the % of situations in which “best” strategies were used in constructing the new path (% of situations for which “best” strategies exist in which they were used).

Now we can rewrite the two premises in the theorem as

(P.1) \( q > .50 \)

(P.2) \( P > C_{pos} / (C_{pos} + C_{neg}) \), \( C_{pos} > 0 \), \( C_{neg} > 0 \)

From our description of the learning mechanism, we know that after a positive comparison

\[ E_{t+1}(V) = E_t(V) + C_{pos} \quad (2.1) \]

and after a negative comparison

\[ E_{t+1}(V) = E_t(V) - C_{neg} \quad (2.2) \]

Hence, the overall expectation following a determinate comparison is

\[ E_{t+1}(V) = P^*(E_t(V) - C_{neg}) + (1 - P)^*(E_t(V) + C_{pos}) \]

but from P.2 we know \( P > C_{pos} / (C_{pos} + C_{neg}) \); therefore

\( (C_{pos} + C_{neg})*P > C_{pos} \)

and from (2.3),

\[ E_{t+1}(V) < E_t(V) \]

In other words, once \( P > C_{pos} / (C_{pos} + C_{neg}) \) we can expect the pairs used in constructing new paths to be decremented. As a result those pairs not used on the path will become more likely to be selected.

Let \( D \) be the expected decrement in the probability of selecting a “best” strategy that was used on the new path.

\( I \) be the expected increment in the probability of selecting a “best” strategy that was not used on the new path.

Since the probabilities of selection in any state must sum to unity, and since there are more than two strategies per state, and only one is decremented,

\[ D > I \] (2.4)

Now we can write an expression for the expected change in the probability of selecting a best strategy.

Let \( \Delta_{prob} \) be the expected increase in the probability of selecting a “best” strategy after a reinforcement.

\[ \Delta_{prob} = -q*D + (1 - q)*I \] (2.5)

but from P.2,

\[ q > .50 \quad (q < 1) \]

\[ q > (1 - q) \]

Therefore, using (2.4), we get from (2.5) that

\[ \Delta_{prob} < 0 \]

Hence, we have shown that the probability of selecting a “best” strategy must decrease, and our theorem is proved.

To test this hypothesis we attempted to train the program again from scratch on the same problems using both positive and negative reinforcement (the decrement for negative reinforcement was 1). The procedure was the same as in Experiment I, but the results were quite different.
TABLE V—A Comparison of Solution Rates in Experiments I and II

<table>
<thead>
<tr>
<th></th>
<th>(1)</th>
<th>(2)*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Mean</td>
</tr>
<tr>
<td>10 Minute Rate</td>
<td>on First Two Learning Trials</td>
<td>on Test Trials</td>
</tr>
<tr>
<td>Experiment I (Positive Reinforcement)</td>
<td>1.6</td>
<td>4.0 +2.2**</td>
</tr>
<tr>
<td>Experiment II (Positive and Negative Reinforcement)</td>
<td>1.9</td>
<td>1.5 -0.4</td>
</tr>
<tr>
<td>Difference (I) − (II)</td>
<td>−0.3</td>
<td>+2.5*</td>
</tr>
</tbody>
</table>

* t = 3.309, df = 10, p < .005
** U = 0, p < .001

Results

The improvement in performance was significantly less for training with both positive and negative reinforcement than it had been in Experiment I for positive reinforcement alone.

In Table 5 the solution rates for training and test trials in Experiments I and II are compared. One can see that the solution rates were significantly inferior when negative reinforcement was included. This is also quite clear from the learning curves shown in Figures 7.1 to 7.3 and 9.1 to 9.2. The performance seems to have improved on the first training trial and then worsened. One can compare the test trials in this experiment with those in the first experiment (Figures 4.1 to 6.2) and note the substantial differences.

The reduction in entropy in the tables of connections trained with negative reinforcement was also less. The reduction in the table trained on Problem 1 was only 13% of the possible. One should not overemphasize this difference, however, since orderliness does not necessarily imply that the desired order has been achieved. Nevertheless, it is interesting to note that 90% of the 13% reduction in entropy during training on Problem 1 occurred during the first training trial. This, of course, is in accord with our hypothesis.

One can see some of the more subtle effects of negative reinforcement more clearly by looking at the within trial behavior of the program. During the three training trials on Problem 1 and the following two trials on Problem 2 there were 518 positive reinforcements of state-strategy pairs in contrast to 1701 negative reinforcements of strategy pairs. Let us look in detail at the behavior of the program within Training Trial 1 on Problem 1 and compare it with the corresponding training trial in Experiment I. The rate of occurrence of negative comparisons was slightly less in this experiment. This is not surprising since negative reinforcements would make it less likely that the program would repeat a series of bad decisions and encounter...
another negative reinforcement. From the large fluctuations in the sum of the entries in the table of connections during the trial, we could see that positive reinforcements were the dominant influence at the beginning of the trial, but that their effect could have been wiped out by the negative reinforcements later in the trial. Besides preventing the repetition of bad decision sequences, negative reinforcement introduces more variety into the decision making process. In other words, negative reinforcement can move the table of connections off a locally optimal structure to search for a better structure. The greater variety in decision making is best seen by comparing Figure 8.2 with Figure 5.2. However, the total effect of these characteristics of negative reinforcement in changing the structure of the table of connections is best shown by Figure 8.1. With negative reinforcement the changes in probability were more erratic. A new strategy suddenly increased in probability after the trial was half over.

This experiment demonstrates that negative reinforcement can never be used as freely as positive reinforcement in learning optimization tasks. The many previous methods based solely on error correction training would perform poorly on optimization tasks. Nevertheless, one can see that negative reinforcement has some desirable effects: it prevents the table of connections from becoming stranded on local optima and causes a greater variety of decisions to be investigated.

Figure 7.3—The solutions produced during test trials on Problems 1 and 3.

Figure 8.1—The probabilities of the program selecting each of the five strategies as a function of the number of comparisons of partial solutions during learning. The data are for one particular task situation (state) corresponding to average time requirements, average criticality, and the beginning of a solution. The data are from the first training trial on Problem 1 in Experiment II.

Figure 8.2—The number of different state-strategy pairings used as a function of the number of comparisons of partial solutions during learning. The data are from the first training trial on Problem 1 in Experiment II.
Hence, we would like to find a way to reap some of the benefits of negative reinforcement while avoiding the pitfalls.

**EXPERIMENT III**

(d) During training should the program always strive to produce the best possible solution?

An implicit assumption in many previous learning mechanisms is that the path to becoming an excellent problem solver is monotonic. However, if this assumption—that the strategies employed by a good problem solver will be worthwhile for an expert—is not always true, then a learning program needs a mechanism for exploring solutions outside of those suggested by its previous learning? In different terms, doesn’t a learning program need a way to escape local optima generated by particular strategies and to experiment with new strategies that may lead to the global optimum?

If one views an optimization problem as the problem of finding an ideal path in a tree of solutions, he can see why a learning program would need such mechanisms. The strategies that generate a path (solution) are reinforced if that path is superior to previous paths. Desirably enough, this somewhat narrows the scope of future search to branches likely to be selected by the same strategies. Eventually, a solution will be reached that cannot be exceeded in a reasonable amount of time. At this point all branches off this path (assuming some entropy in the search process) and all branches likely to be reached with the same strategies should have been tried and found inferior. But there is no guarantee that a radical change in several strategies at some point on the path might not lead to an equal or better path. The danger is that a few radical changes in strategies might consistently produce quite different paths and superior solutions, but these changes would never be investigated because any one of them, alone, coupled with the learned series of strategies, only leads to a branch off the old path and a worse solution. Therefore, it is suggested that a program that adopts a short period of relatively non-directive search at the end of a learning sequence where improvement has terminated will learn a superior decision structure and eventually perform better than a program that spends all its time searching on the basis of its past experience.

Admittedly, such a non-directive search would be time consuming and costly in that performance would be bad during learning. MacKay, in fact, has suggested deliberately selecting bad strategies during learning so that they can be eliminated with negative reinforcement. However, such a method would not help much in selecting a strategy of little utility most of the time that is of the highest utility in moving from good to excellent solutions. It is suggested that what is needed is a mechanism for relatively random exploration of strategies whenever it appears that the program is “hung up” on a local optimum. How useful such an addition to a learning procedure would be is the fourth issue for study.

![Figure 9.1](https://www.computerhistory.org)

*Figure 9.1—The solutions produced during training on Problem 3 with positive and negative reinforcement.*

**Figure 9.2**—The solutions produced during test trials on Problems 1 and 3.

From the collection of the Computer History Museum (www.computerhistory.org)
BASE SOLUTION

Figure 10.1—The solutions formed during special additional training on Problem 1 in Experiment III. The initial memory structure has been trained with positive reinforcement alone in Experiment I (see Figs. 4.1 and 4.2). Both positive and negative reinforcement were used on these three trials. In addition, after the first benchmark was formed, strategies were selected completely at random during Trials 1 and 3.

We began with the final table of connections from training on Problem 1 in Experiment I. It appeared that this table had become stranded on a local optimum. Although it produced good solutions, better solutions existed that it could not find. Additional training did not help since improvement is needed for positive reinforcement to be applied. Hence, we decided to give the table of connections additional training with negative reinforcement and with strategies selected at random. With this scheme good strategies would be no more likely to be selected than bad ones, and negative reinforcement should not destroy the table of connections; rather it should move the table off the local optimum and allow the learning program to search for another optimum.

Results

The results of the experiment indicate that this is exactly what happened. Two random strategy selection trials were combined with one normal training trial. These are shown in Figure 10.1. During the first trial,
where strategies were chosen at random, negative reinforcement altered the table of connections moving it off its local peak. The second trial, a normal training trial, established new connections in the table with positive reinforcement. The third trial again varied the table with negative reinforcement and random strategy selection. Finally, on a test trial (Figure 10.2) the performance program produced good solutions to all three problems and the best solutions to Problems 1 and 2 that were ever generated. The final table of connections is displayed in Table 6.

A study of the within trial behavior of the program
Figure 11.2—The number of different state-strategy pairings used as a function of the number of comparisons of partial solutions during learning. The data are from the first training trial on Problem 1 in Experiment III, a trial during which strategies were selected at random.

confirms these effects. Comparing Figure 11.2 with Figures 5.2 and 8.2 indicates that a greater number of state-strategy pairings were investigated in this learning mode than in the other modes. The average value of cells in the table of connections dropped rapidly, making the table more susceptible to new applications of positive reinforcement. At the same time this random selection mode did not destroy all the previous learning. The random selection trial actually increased the superiority of strategy S25 in the state shown (Figure 11.1).

From these results one can conclude that it is not always the best policy during learning to try and find good solutions. At some times one is better off investigating new state-strategy pairings and ignoring the immediate consequences.

CONCLUSIONS

The study demonstrated that a program for solving a very complex optimization task (a scheduling task) could learn generalizable performance principles solely through ordinal feedback from intercomparisons of the solutions it had produced. In particular connections were formed from a very restricted state description of the environment to basic performance strategies. Only a relatively small number of positive reinforcements were necessary to produce significant improvement. Furthermore, and perhaps most important of all, negative reinforcement or error correction training was shown to be a hindrance to learning when only ordinal feedback was available. Theoretical evidence was provided in support of this empirical finding. Finally, the study established that some benefits can be derived from negative reinforcement by using it with a performance routine that does not always preform at its best.

This study did not address (directly) the problem of how the program could generate the stimulus features and basic strategies that are the entries in the table of connections. However, a table of connections provides a great deal of information that can be used in eliminating old entries and introducing new entries. Some simple descriptive statistics about a table, e.g., variance of subsets of entries, provide information on just what basic strategies or features should be eliminated. Several algorithms have already been developed to make use of this information.

Finally, this study raises some very interesting questions about human learning ability. Do humans sometimes ignore the consequence of their behavior in order to learn more rapidly? What is the relative power of ordinal, interval, and ratio feedback in learning complex tasks? Do humans change their state-strategy pairings on the basis of failures to improve? How large a set of state variables do humans attend to at any one time?

REFERENCES

1 M MINSKY S PAPERT
   Perceptrons
   MIT Press Cambridge Massachusetts 1969
2 N J NILSSON
   Learning machines
   McGraw-Hill New York 1965
3 D B YNTEMA, G E MUESER
   Keeping track of variables that have few or many states
4 H FISHER G L THOMPSON
   Probabilistic learning combinations of local job-shop scheduling rules
   Industrial Scheduling (eds M Muth and G L Thompson)
   Prentice-Hall Englewood Cliffs 1963
5 A NEWELL J C SHAW H A SIMON
   A variety of intelligent learning in general problem solver
   Self-Organizing Systems (eds M Yovitts and S Cameron)