CS 118
A. I. 74

AD 681027

MACHINE LEARNING OF HEURISTICS

BY

DONALD ARTHUR WATERMAN

SPONSORED BY
ADVANCED RESEARCH PROJECTS AGENCY
ARPA ORDER NO. 457

DDC
JAN 22 1969
A

DECEMBER 1968

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

Reproduced by the
CLEARINGHOUSE
for Federal Scientific & Technical
Information Springfield Va. 22151

This document has been approved for public release and sale; its distribution is unlimited

BEST
AVAILABLE COPY

Machine Learning of Heuristics

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE

AND THE COMMITTEE ON THE GRADUATE DIVISION

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

By

Donald Arthur Waterman

December 1968

STANFORD ARTIFICIAL INTELLIGENCE REPORT
MEMO NO. AI-74

December 1968

MACHINE LEARNING OF HEURISTICS

by Donald Arthur Waterman

ABSTRACT: First, a method of representing heuristics as production rules is developed which facilitates dynamic manipulation of the heuristics by the program embodying them. This representation technique permits separation of the heuristics from the program proper, provides clear identification of individual heuristics, is compatible with generalization schemes, and expedites the process of obtaining decisions from the system.

Second, procedures are developed which permit a problem-solving program employing heuristics in production rule form to learn to improve its performance by evaluating and modifying existing heuristics and hypothesizing new ones, either during a special training process or during normal program operation.

Third, the abovementioned representation and learning techniques are reformulated in the light of existing stimulus-response theories of learning, and five different S-R models of human heuristic learning in problem-solving environments are constructed and examined in detail. Experimental designs for testing these information processing models are also proposed and discussed.

Finally, the feasibility of using the aforementioned representation and learning techniques in a complex problem-solving situation is demonstrated by applying these techniques to the problem of making the bet decision in draw poker. This application, involving the construction of a computer program, demonstrates that few production rules or training trials are needed to produce a thorough and effective set of heuristics for draw poker.

The research reported here was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense (SD-183).

ACKNOWLEDGMENTS

I wish to express my sincere thanks and appreciation to my principal thesis advisor, Professor Edward A. Feigenbaum, not only for his perceptive guidance, his intellectual inspiration, and his discerning criticisms, but also for the friendly encouragement and moral support he so generously provided. I am also grateful to Professor Gordon H. Bower for the extensive time and effort he spent enlightening me with regard to the psychological aspects of my thesis topic.

In addition I am indebted to Dr. Bruce G. Buchanan, Professor D. R. Reddy, and Professor David J. Gries for their valuable suggestions and critical evaluations of this thesis.

I would also like to thank Mrs. Grace Mickelson and Mrs. Gail Schwartz for their excellent job in typing and proofreading this report, Mrs. Dorothy McGrath for her fine illustrations, Miss Dianna Konrad for supervising the preparation of this report, and Miss Barbara Chiarle for reproducing parts of this report.

BLANK PAGE

TABLE OF CONTENTS

CONTENTS (Continued)

LIST OF ILLUSTRATIONS

ILLUSTRATIONS (Continued)

Table

CHAPTER 1

HEURISTIC PROBLEM-SOLVING BY COMPUTER

1.1 INTRODUCTION

Currently much research is being done with computers in an attempt to produce programs which exhibit intelligent behavior. This work can be divided into two main categories, (1) artificial intelligence research, and (2) research in the simulation of cognitive processes (Feigenbaum and Feldman, 1963). The former is concerned with programming computers to perform intellectual tasks, while the latter is concerned with programming computers to simulate human cognitive processes.

The goal of artificial intelligence research is the construction of computer programs which exhibit intelligent behavior, with the emphasis placed on the degree of intelligence exhibited. The goal of research in the simulation of cognitive processes, on the other hand, is the construction of computer programs which simulate human cognitive behavior, with the emphasis placed on the degree to which the programs can predict this behavior.

To illustrate the distinction between these two categories consider the intellectual task of game playing. A researcher in artificial intelligence would judge the merits of his game-playing program on the basis of its skill at playing the game, the ideal program being one capable of defeating all other players. However, a researcher in the simulation of cognitive processes would base the evaluation of his game-playing program on the extent to which its game decisions or "moves" paralleled those of human players, not on how well his program played the

game. This distinction is not a clear one, since some research efforts can be classified as belonging to both categories. One example of this is the NSS Chess Player (Newell, Shaw, and Simon, 1958), a program, proficient at playing chess, which employs many human-like problem-solving techniques.

In both the artificial intelligence area and the simulation of cognitive processes area extensive use is made of heuristic programming, that is, of employing heuristics in programs which solve complex problems. The utility of most of these heuristic programs depends to a large extent on the form or character of the heuristics employed. Thus heuristics play an important role in the attempt to create programs which exhibit intelligent behavior.

One of the important unsolved problems of artificial intelligence research today is that of the learning of heuristics (Feigenbaum and Feldman, 1963). The question is this: how can computers (and how do people) learn new heuristic rules and methods which can be used to facilitate decision-making in a problem-solving situation? Furthermore, how are these new heuristics combined with existing ones to produce a functional system capable of intelligent decision making? Solutions in this problem area, besides permitting the construction of very powerful problem-solving programs might also suggest what direction psychological theories of learning should take. This paper will be concerned primarily with the development of computer programs which learn heuristics in a problem-solving environment.

1.2 DEFINITION OF HEURISTIC METHODS

In this section the concept of the heuristic will be discussed in detail. First, the term "heuristic" will be informally defined and contrasted with the concept of the algorithm. Next, more formal definitions of these terms will be presented, and the implications of these definitions examined.

Informal Definitions

A heuristic (heuristic procedure, heuristic method) is a rule-of-thumb, strategy, trick, simplification, or any other kind of device which drastically limits search for solutions in large problem spaces (Feigenbaum and Feldman, 1963). A heuristic does not guarantee a solution, rather it supplies solutions which are acceptable most of the time. On the other hand, an algorithm (from the logician's viewpoint) is any set of operations which can be represented by a Turing machine (Trakhtenbrot, 1963). However, when "algorithm" is contrasted with "heuristic" a narrower definition is usually implied. In the narrow sense an algorithm is a well-defined search procedure which is guaranteed to produce the correct solution, given enough time. The advantage in using a heuristic method rather than an algorithmic one is often that of reduced search time and effort. The disadvantage is that a solution may not be found, and if one is found it may not be optimal.

EVALUATION. The above informal definitions give a clear, intuitive picture of what is usually meant by the term "heuristic" but are unsatisfactory in two respects. First, these definitions lead to much confusion concerning the nature of the differences between heuristic and algorithmic

methods. For example, they fail to provide the answers to the following questions:

(1) Can a search procedure be both heuristic and algorithmic?

(2) Does a heuristic procedure necessarily imply failure on some problems?

(3) How does one show that a given procedure is a heuristic one? An algorithmic one?

Confusion concerning these and related questions has led to a good deal of controversy in this area.

Second, these definitions state that a heuristic necessarily implies reduced search time or effort in a problem area, thus denying the existence of heuristics which do not lead to reduced search time or effort. This constraint leads to definitions which are satisfactory for the typical heuristic problem-solving program; i.e., one where the heuristics are embedded in the program and can be changed only by some external operation, such as the programmer revising portions of the code. However, these definitions are not satisfactory for the type of program to be described in this paper, a program which hypothesizes, evaluates, and modifies its own heuristics. For this type of program the concept of a "poor" (inadequate, ineffective, or useless) heuristic is needed since the program itself must be able to determine whether a given heuristic is a "good" or "poor" one; and thus decide whether to retain it or discard it. It cannot be assumed that every procedure hypothesized by this type of program will lead to reduced search time or effort, but it would be convenient to think of all these procedures as heuristics. This can be accomplished if the definition of the term heuristic carries no stipulation about search time or effort but instead uses the search

time or effort as one of the criteria for the "goodness" or "worth" of the heuristic.

Formal Definitions

In this paper the terms computational rule, algorithm, and heuristic will be taken to mean the following.

Computational Rule: any procedure determined by a set of instructions that specify at each moment precisely and unambiguously what is to be done next.

Algorithm: a computational rule which obtains solutions to problems, such that there exists at least one problem domain where for every problem in the domain this computational rule produces the correct solution. Furthermore, the computational rule is said to be an "algorithm for" each problem domain satisfying the above requirement.

Heuristic: a computational rule which obtains solutions to problems, such that there exists at least one problem domain where the computational rule obtains one or more correct solutions but where it is not true that the computational rule will produce the correct solution for every problem in the domain. Furthermore, the computational rule is said to be a "heuristic for" each problem domain satisfying the above requirement.

These formal definitions satisfy the two conditions that the informal definitions failed to satisfy. That is, (1) providing a clear distinction between heuristic and algorithmic methods, and (2) admitting the existence of heuristics which fail to lead to reduced search time or effort.

IMPLICATIONS. From the formal definitions given above it is clear that for any computational rule, CR, and problem domain, D, if CP produces any correct solutions in D then it is always true that CR is either a heuristic for D or an algorithm for D, but never both. However, a computational rule may be both a heuristic and an algorithm; for example, CR might be a heuristic for problem domain D1 but an algorithm for domain D2. Also, it is possible that a computational rule could be a heuristic for more than one problem domain.

To show that a computational rule CR is an algorithm for a problem domain D one must

(1) show that CR produces the correct solution for every problem in D.

To show that a computational rule CR is a heuristic for a problem domain D one must

(1) show that CR produces a correct solution for a least one problem in D.

(2) show that CR fails to produce a correct solution for at least one problem in D.

It should be noted that under these formal definitions, a heuristic procedure does necessarily imply failure on some problems.

If one is unable to show that a particular computational rule CR (which produces correct solutions in problem domain D) is an algorithm for D, and is also unable to show that CR is a heuristic for D then the status of CR is unknown, although it is still either an algorithm or a heuristic (but not both) for D. Since the members of this class of computational rules are generally thought of as being heuristics, in this paper they will, for convenience, be labeled or "hypothesized"

as heuristics with the understanding that their status is actually unknown and may be discovered or proven at some later date.

HEURISTIC PROGRAM. A program will be considered to be a computational rule precise enough to be executed by a computer, and a heuristic program simply a program which contains heuristics. Thus under the formal definitions given, a heuristic (or heuristic procedure) is just a heuristic program containing exactly one heuristic. And conversely a heuristic program is actually a heuristic for some particular problem domain. Figure 1-1 illustrates how a heuristic program for chess (Bernstein and Roberts, 1958) could be considered a heuristic for the problem domain D1 while containing heuristics for domains D2 , D3 , D4 , and D5 .

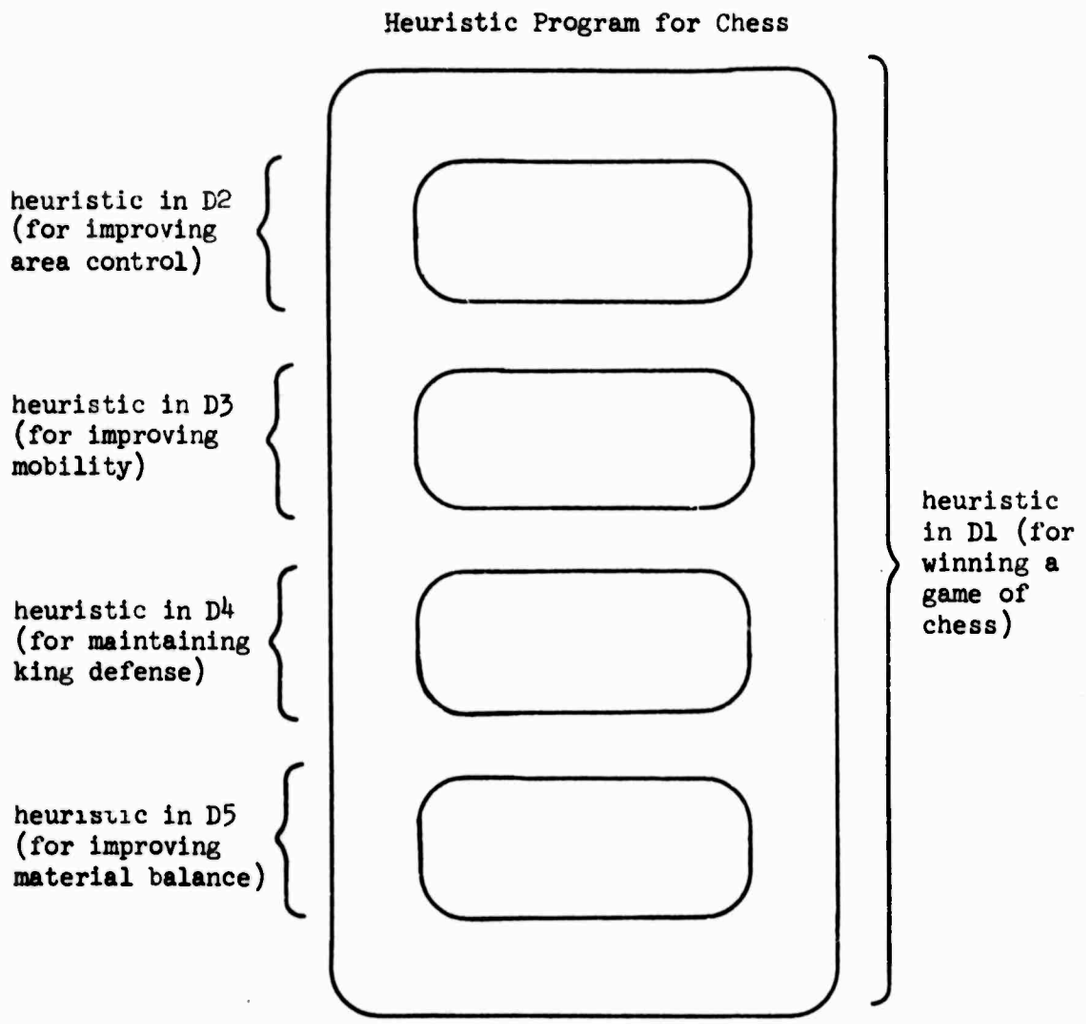


Figure 1-1. Structure of a heuristic program for chess, illustrating how the program is a heuristic for domain D1 while containing heuristics for domains D2, D3, D4, and D5.

HEURISTIC POWER. The usefulness or "power" of a heuristic (as formally defined) is dependent on two criteria:

(1) the search time or effort involved in obtaining a solution, and

(2) the percentage of problems in the domain which can be correctly solved.

A very useful, good, or powerful heuristic would thus be one requiring only a short search time to find a solution, while having the capability of correctly solving a large percentage of the problems in the domain. On the other hand, the usefulness of an algorithm is dependent on just one criterion, the search time or effort involved in obtaining a solution. The percentage of problems correctly solved is not relevant since by definition the algorithm always solves all the problems in the domain. These criteria are demonstrated graphically in Figure 1-2 (Anonymous, 1967). Here algorithm A_1, is unequivocally superior to heuristic H_1, algorithm A_2, and heuristic H_2; i.e., $A_1 > H_1, A_2, H_2$. In the 0-3 hour range $H_1 > A_2 > H_2$, but in the 0-5 hour range $A_2 > H_1 > H_2$, and in the 0-7 hour range $A_2 > H_2 > H_1$. This clearly illustrates how a heuristic can prove more useful than an algorithm when the search time or computing effort is restricted, since H_1 is superior to A_2 when the computing effort is limited to 3 hours or less.

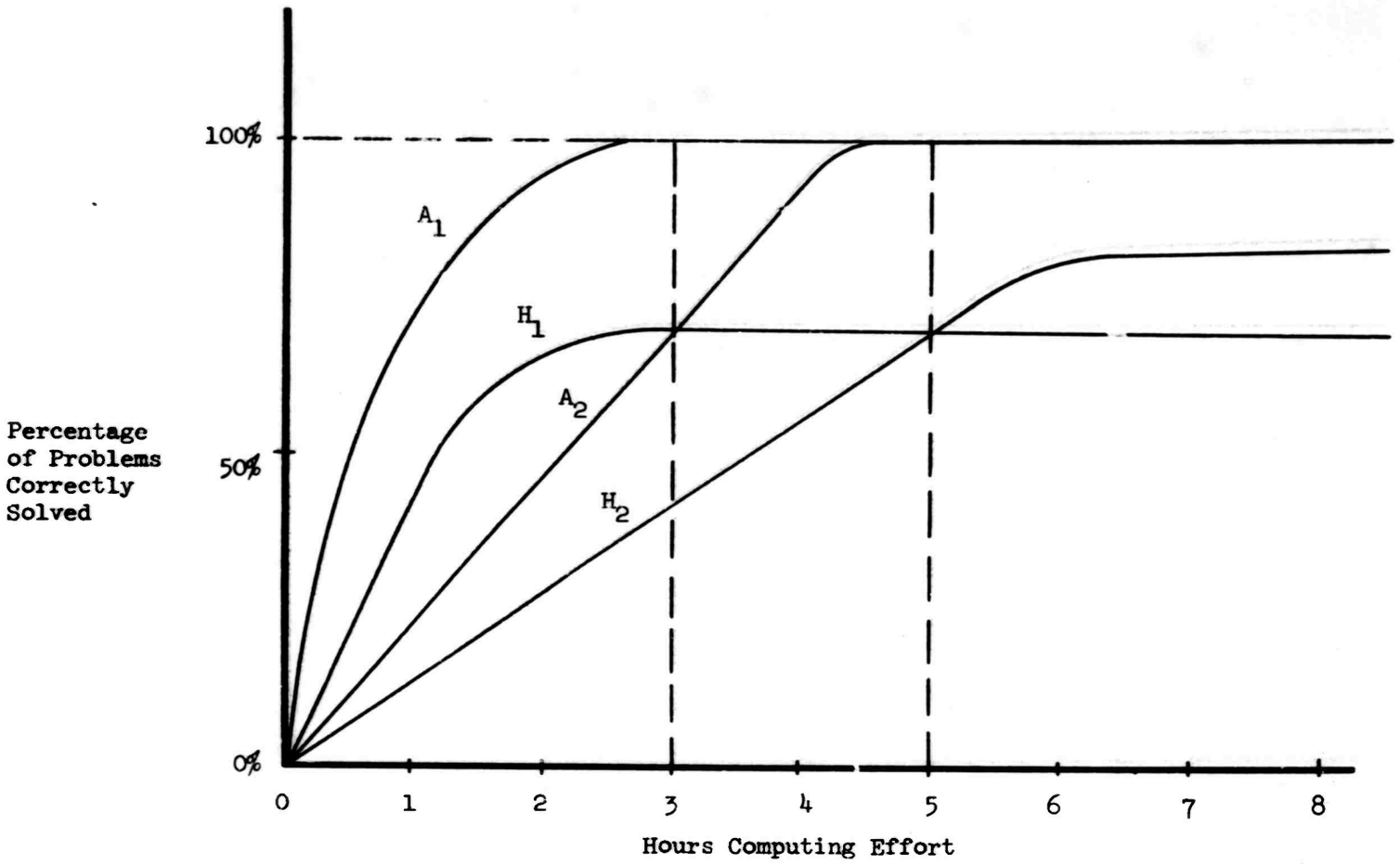


Figure 1-2. Graphical illustration of the criteria for the usefulness or power of heuristics.

1.3 HISTORICAL BACKGROUND

In the last decade a large number of computer programs employing heuristics have been written, most of them being of a nonnumerical nature. Some of the more important programs of this type will now be briefly discussed. For this discussion it will be convenient to think of them as being divided into two categories: (a) programs designed primarily to demonstrate problem solving techniques, such as game playing, theorem proving, and question answering, and (b) programs designed primarily to demonstrate learning techniques, such as pattern recognition, concept learning, and verbal learning.

Problem Solving Programs

LOGIC THEORIST. One of the landmarks in the development of heuristic programming is a program written by Newell, Shaw, and Simon which attempts to prove theorems in elementary logic. (Newell, Shaw, and Simon, 1956, 1957a, 1957b; Stefferud, 1963). This program, called the Logic Theory machine (or LT), uses heuristic methods to discover proofs in the Russell-Whitehead system for the propositional calculus.

Initially, the program is given a set of axioms to use and the problem of finding a proof for a particular theorem. The program first tries the method of substitution on the theorem; that is, LT compares the theorem with each axiom to see if through substitution of free variables and connectives the theorem can be made to match one of the axioms, thereby solving the problem. If no match can be found a number of subproblems are generated, each being the task of proving valid a particular proposition whose validity implies the validity of the original theorem. The method of substitution is then tried on the

subproblems and if no match can be found subproblems of each subproblem are generated and the procedure is again applied to each of them. The search continues in this fashion until a solution is found or the program runs out of time.

Some of the important heuristics used in LT include (1) the heuristic technique of working backward from the theorem to be proved toward the axioms, (2) the methods used to generate subproblems, and (3) the heuristics for deciding which subproblem out of a group of subproblems should be attempted first (i.e., which subproblem is easiest to solve) and which should not be attempted at all. The heuristics used in LT are an integral part of the program and are thus difficult to recognize and specify precisely.

The LT project has been criticized (Wang, 1960a) on the grounds that there exist mechanical decision procedures for the propositional calculus which will find the proof of any valid theorem and will find it faster than does LT. Minsky (1961) answers this criticism by noting that the purpose of LT is primarily to study techniques for solving difficult problems rather than to produce an expert theorem proving program in the propositional calculus. The techniques used by LT can be applied to many different problem areas, whereas Wang's decision procedure is applicable only to the propositional calculus. This is not meant to imply that decision or proof procedures are of little importance in artificial intelligence; much progress has been made, for example, in the area of proof procedures for the predicate calculus (Wang, 1960b, 1961; Davis and Putnam, 1960; Davis, 1963; Robinson, Wos, and Carson, 1964; Wos, Carson, and Robinson, 1964; Robinson, 1965; Slagle, 1967).

LT APPLICATIONS. The techniques used by LT have been successfully applied to a number of different problem areas. A program for proving theorems in plane geometry (Gelernter, 1959; Gelernter, Hansen, and Loveland, 1960) has been developed which starts with the theorem to be proved and like LT generates subproblems in an attempt to work backward toward one of the given axioms. Elementary symbolic integration problems have been solved using this same general approach. (Slagle, 1961). Here the program starts with an expression to be integrated (main problem) and generates other expressions to be integrated (subproblems) such that the solution of certain subproblems leads to the solution of the main problem. A subproblem is solved (expression integrated) when the expression can be made to match one of a set of standard forms whose integrals are known. These standard forms are thus analogous to the axioms of the Logic Theory machine.

Another example of the LT influence can be found in the area of question answering programs. A program has been written (Black, 1964) which is designed to answer questions put to it in advice-taker notation (McCarthy, 1959) by working backward from the question, generating subquestions, in an attempt to match these subquestions with given statements known to be true. Recently, work has been done on incorporating the LT techniques into a general purpose program capable of constructing proofs for propositions in a number of different problem domains (Slagle and Bursky, 1968).

GENERAL PROBLEM SOLVER. Out of the Logic Theory machine grew a more powerful program called the General Problem Solver (GPS), designed to simulate human problem-solving processes (Newell, Shaw, and Simon, 1959; Newell and Simon, 1961). This program deals with a task environment consisting of

objects and operators. The problem is usually of the form "given an initial object A and a desired object B , find a sequence of operators, S:Q_1, ...Q_k , that will transform A into B ". In this formulation the problem is one of heuristic search, a process which underlies much of the recent work in problem solving programs (Newell and Ernst, 1965). To solve this problem GPS has three types of goals available:

(1) Transform object A into object B ,

(2) Apply operator Q to object A ,

(3) Reduce the difference D between object A and object B .

Associated with each goal is a set of methods related to achieving goals of that type. Hence solving the problem consists of selecting an appropriate goal, evaluating this goal in context to see if it is worth attempting, and executing the methods associated with the goal, if the goal deemed feasible. If the methods include achieving one or more of the three goals just described then these are considered subgoals whose attainment leads to the attainment of the initial goal. GPS attempts to solve the problem of transforming A into B by generating, in a "depth first" fashion (Newell, 1962), goals and subgoals relevant to reducing the differences between A and B .

One of the initial applications of GPS has been to the problem of proving theorems in the propositional calculus. For this particular task, the objects are logic expressions, the operators are axioms or rules for transforming one logic expression into another, and the differences between objects which are recognized by the program include features like the logical connectives employed or the number of occurrences of a variable. Besides being given the definitions of the objects, operators, and differences, the program must also be supplied with a

connection table which associates with each difference a set of operators relevant to modifying that difference. Once the task environment is so defined, GPS is ready to attempt to prove theorem A, a logic expression in the propositional calculus, by transforming it into a given expression B which is a known axiom in the propositional calculus.

The important heuristics used in GPS are (1) those connected with the methods used to try to achieve the generated subgoals, (2) heuristics for deciding whether or not a particular subgoal is worth attempting, and (3) the technique of planning, i.e., constructing a tree of subgoals based on an abstracted problem space composed of simplified objects and operators, and then using this tree as a plan of attack for the actual problem space of complex objects and operators. Most of these heuristics deal directly with the manipulation of objects and differences. In contrast, the heuristics of LT deal with the manipulation of theorems and axioms in the propositional calculus. It is precisely this difference that makes GPS a "general" problem solver, that is, capable of solving problems in any domain where the problem can be specified in terms of objects, operators, and differences.

Besides proving theorems in logic, GPS has also been used to solve trigonometric identities (Newell, Shaw, and Simon, 1959). Programs employing GPS problem solving techniques have been written which balance assembly lines (Tonge, 1961), compile computer programs (Simon, 1961, 1963), and simulate human behavior in the binary choice experiment (Feldman, Tonge, and Kanter, 1963).

CHESS-PLAYING PROGRAMS. Game playing is another area which is quite

amenable to the development of heuristic programs. In this area, a large portion of the work has been concentrated on the development of programs for playing chess. Shannon in 1949 proposed a framework for a chess playing program which in essence stated that (1) the chess game can be thought of in terms of a game tree whose nodes correspond to board configurations and whose branches correspond to the alternative legal moves and, (2) the best move to make from a particular node N1 (i.e., in a particular board situation) can be determined by generating alternative moves in the tree down to some particular depth, evaluating the board configurations at that depth as single numerical values, and minimaxing (Slagle, 1963) these values back up the tree to node N1 , picking from N1 the alternative move which received the highest value (Shannon, 1950; Newell, Shaw, and Simon, 1958).

Turing has described a program based on Shannon's proposal which, in determining the best move, generates all possible alternative moves down the tree until a dead position with regard to piece exchange is reached at each branch (Turing, 1950). A group at Los Alamos has programmed MANIAC I to play chess, also generating all possible alternative moves but only down the tree to a fixed depth of 4 moves (Kister et al., 1957). The program performs only a minimal evaluation of the board configurations at this depth, before minimaxing to determine the best alternative. A program written by Bernstein plays chess using this same framework but generates only 7 plausible alternatives at each node down to a fixed depth of 4 moves, where it performs an extensive evaluation of the board configuration before minimaxing (Bernstein and Roberts, 1958).

NSS CHESS PLAYER. Newell, Shaw, and Simon have developed a chess program

which differs in a number of respects from the programs just described (Newell, Shaw, and Simon, 1958). A set of goals are defined (king safety, material balance, etc.) and alternative moves are generated which tend to satisfy the top priority goals in the given situation. The tree is generated until at each branch a dead position is reached with respect to all goals, that is, until no move can be made which will drastically alter the situation with respect to these goals. The board configuration at each dead position is then evaluated as a list of values (one for each goal) describing how well that configuration meets each goal, and these lists are minimaxed back up the tree. An alternative move is chosen as being a satisfactory one if the list associated with it through minimaxing is greater, element by element, than a list representing the minimum allowable values for each goal.

The important heuristics used in the chess programs just described are (1) those concerned with the generation of alternative moves, (2) those concerned with the depth of analysis, and (3) heuristics for the evaluation of board configurations. Again it is difficult to recognize and specify precisely the heuristics used by these programs, since they tend to be interrelated and are an inseparable part of each program.

Learning Programs

PATTERN-RECOGNITION PROGRAMS. Pattern-recognition research has led to the development of many programs which employ learning mechanisms. Much of the initial work in pattern recognition was based on neural network learning techniques (Carne, 1965), the most successful example of these techniques being Rosenblatt's perceptron (Rosenblatt, 1958, 1962; Green, 1963). The perceptron is basically a network of randomly inter-connected neural

elements, each element being capable of "firing" or putting out a fixed amplitude signal over its output connection lines whenever the sum of the signals on its input connection lines exceeds some threshold. The network learns through reinforcement procedures, the most common type consisting of presenting the network with a stimulus (a set of input signals) and for each learning trial incrementing the output amplitude of all elements which fire when the correct response (output signal) is made.

A more sophisticated pattern-recognition model, Pandemonium (Selfridge, 1959), uses a highly organized network where the elements represent likely features of the input patterns. The model learns by adjusting the weights associated with the connections between these elements and the possible responses. For example, if the model were given a pattern containing feature f_1 and was told that the pattern belonged in class R_1, then the weight on the connection between element f_1 and response R_1 would be incremented, meaning that a pattern with feature f_1 would then have a greater probability of being classified as type R_1. One problem with this type of model is that the features it uses must be supplied to it by the designer, and it is seldom clear what features will lead to efficient operation. A pattern-recognition program has been written (Uhr and Vossler, 1961), which attempts to overcome this difficulty by effectively generating features at random, evaluating them in terms of their usefulness, and discarding those which are not useful. The program not only learns to classify patterns by adjusting weights or coefficients on the features, but also learns what features can be used to classify the patterns.

In the pattern-recognition programs just described the learning consists essentially of using a reinforcement process as the basis for generalizing by adjusting weights or coefficients. The heuristics involved include those connected with the determination of features to use and those concerned with the techniques used to adjust the weights.

SAMUEL'S CHECKER-PLAYING PROGRAM. One of the most successful learning programs to date is a checker-playing program which learns to improve its playing ability through training and game-playing experience (Samuel, 1959, 1960). This program is patterned after the framework proposed by Shannon for the game of chess. As in the chess programs described earlier, the checker program bases its move decision on the results of looking ahead in the game tree to relatively dead positions, evaluating the board configurations at these positions, and minimaxing these values back up the tree. The value of a board configuration is determined by calculating the numerical value of a linear scoring polynomial $w_1f_1 + w_2f_2 + \ldots + w_nf_n$, where the f's represent certain parameters or features of the board configuration (such as piece advantage, denial of occupancy, mobility, and center control) and the w's are weights or coefficients representing the relative importance of each parameter.

The checker program is capable of two basic types of learning, (1) rote learning and (2) generalization learning. The rote learning is quite elementary and consists of storing in memory all the board positions encountered during play together with their scores based on lookahead minimaxing. Performance improves under this learning scheme since the program saves time when it encounters familiar board positions, and this time can be used for searching the game tree to a greater depth.

The generalization learning, on the other hand, is somewhat complex and involves adjusting the coefficients of the scoring polynomial toward their optimal values.

BOOK LEARNING. In one form of generalization learning the program is "trained" by being given a large number of board positions and the associated book moves (the moves recommended by master checker players). During this book learning procedure the program keeps track of the parameters whose values have a general tendency to increase as a result of the book moves and also those whose values have a tendency to decrease. The parameters whose values increase are considered to be important for winning the game and their coefficients are incremented. Conversely, the parameters whose values tend to decrease are considered unimportant and have their coefficients decremented.

LEARNING THROUGH GAME PLAY. In another form of generalization learning the program modifies the coefficients during actual play by comparing, (for each of its moves) the backed-up score for the board position with the score calculated directly from the scoring polynomial. It is assumed that the backed-up score is more accurate than the direct score, hence the coefficients of the parameters are adjusted so that the direct score will more nearly approximate the backed-up score. Parameters which have a general tendency to increase the difference between the backed-up and the direct scores are removed from the polynomial and replaced by parameters from a reserve list. Thus the program can radically modify its evaluation polynomial and can possibly learn which of a given set of parameters are relevant to the goal of winning at checkers.

SIGNATURE TABLES. One difficulty with implementing learning by adjusting coefficients in a linear polynomial is that there exists in this procedure an implicit assumption of independence of the parameters involved, while in actual fact the parameters are seldom independent. Samuel (1967) has proposed a "signature table" scheme to help overcome this problem. In its simplest form this scheme consists of grouping the parameters into sets called signature types, and for each set defining a function which when given a value for each parameter of the set generates a number reflecting the relative worth of that particular combination of parameter values. Each function is defined by enumeration; that is, by a table pairing each combination of parameter values with a number indicating their worth. To keep the tables small the range of parameter values is restricted to either 3, 5 or 7 values. A board position is then evaluated by evaluating each signature table using the parameter values of that position and adding together the numbers obtained from each table. The signature table approach proves to be more efficient than the linear polynomial method when book learning is employed.

In the checker program, learning consists of generalizing by modifying coefficients of board parameters. Among the heuristics used are those concerned with depth of analysis, tree pruning techniques (such as the alpha-beta procedure: Slagle, 1963; Samuel, 1967), determination of parameters, specification of the evaluation function, and the adjustment of coefficients. Heuristics which are used but are seldom acknowledged in this type of program are those connected with the definitions of the parameters; for example, mobility can be defined in many ways, but one definition is likely to be more useful than the

others. The particular definition chosen can be considered a heuristic for measuring the value of the parameter.

CONCEPT-LEARNING PROGRAMS. Programs have also been written which simulate human learning processes. One of the important contributions in this area is a concept-learning program by Hunt (1962, 1966) which learns to distinguish between positive and negative instances of a concept after it is presented with a small sampling of positive and negative instances. Hunt represents an instance of a concept as a set of attribute values, for example, (LARGE, RED, TRIANGULAR) is a positive instance of the concept "large triangle", while (LARGE, RED, CIRCULAR) and (SMALL, RED, TRIANGULAR) are negative instances. The learning process consists of growing a decision tree whose nodes represent tests on the attribute values, such as "is the object large?" or "is the object triangular?". The decision tree is used to classify any given instance as being either positive or negative by sorting the instance down the tree to a terminal node and assigning the instance to the category associated with that terminal node.

To illustrate this process consider the sampling of positive and negative instances given in the above example for the concept "large triangle". The program would use these instances to grow the following tree.

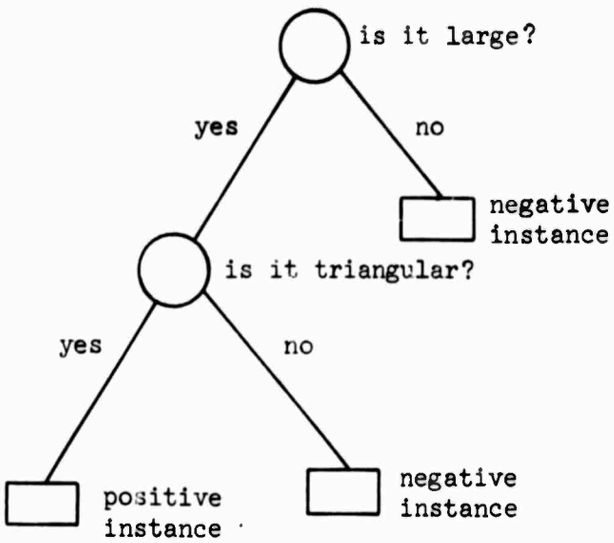


Figure 1-3.

It is clear that if a new instance, such as (LARGE, BLUE, HEXAGONAL) is presented it will be sorted to the proper terminal node (negative, in this case) and thus correctly identified. Another program which performs concept learning is one written by Kochen (1960, 1961). This program, like Hunt's, generates a decision rule for deciding whether or not a given object belongs to a certain class, but makes no attempt to simulate human behavior.

In the concept-learning programs the process of learning consists of making clever generalizations based on the given information. The important heuristics used in Hunt's program are those concerned with the choice of attribute values to use as tests for the nodes and the order in which the chosen values are arranged in the tree.

SIMULATION OF VERBAL LEARNING. Another important contribution in the area of simulation of human learning is a program called EPAM (elementary

perceiver and memorizer), which simulates verbal learning behavior by memorizing three-letter nonsense syllables presented in associate pairs or serial lists (Feigenbaum, 1959, 1963, 1964, 1967). EPAM's task for each pair of syllables S,R is to learn to produce the response R when given the stimulus S . The program accomplishes this by growing a discrimination net composed of nodes which are tests on the values of certain attributes of the letters in the nonsense syllables. For example, a test at one node might be "does the third letter of the syllable have a horizontal component?". The various stimuli and responses are individually sorted down the net to terminal nodes where they are stored, one per terminal node. If two different syllables are sorted to the same terminal node a new test node is grown at that point capable of distinguishing between the two syllables and thus sorting them into two separate terminal nodes. In this fashion the discrimination net is grown. A complete description (all 3 letters) of each response is stored in the net, but for each stimulus only a partial description (1 or 2 letters) is stored together with a cue or partial description of the associated response.

As an illustration of this process consider the task of learning the two pairs of syllables, RAX - JIF and JEQ - HOX. The program would grow the following type of net.

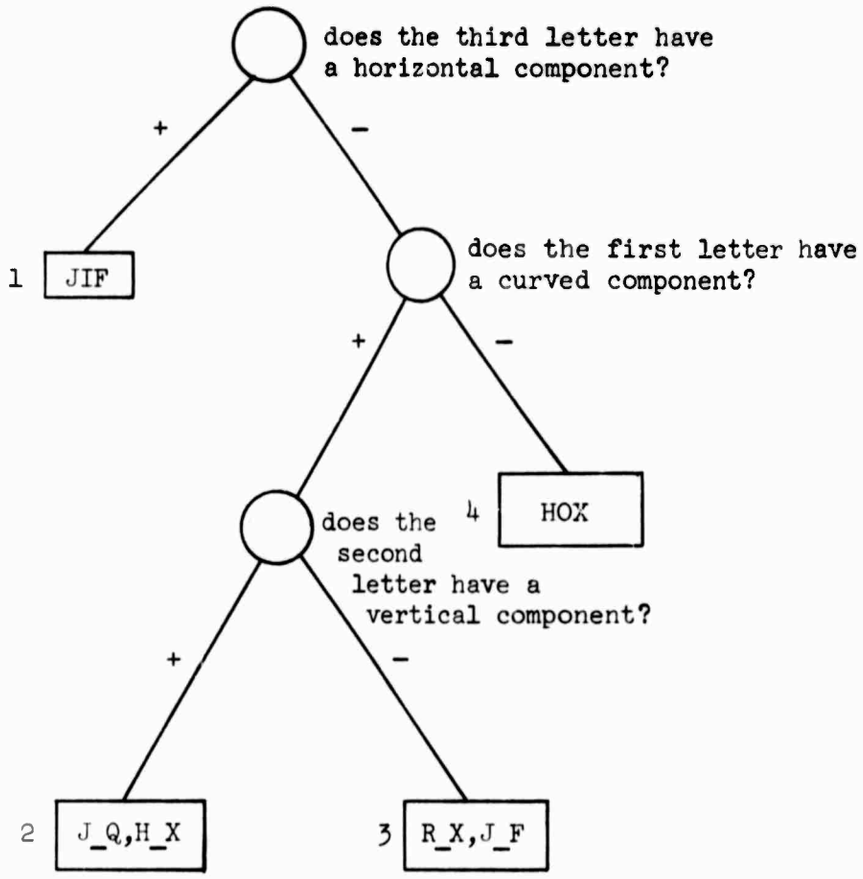


Figure 1-4.

Now if EPAM is given RAX and asked for the response, it sorts RAX down to terminal node 3, retrieves the cue J_F , sorts it down to terminal node 1 and responds with JIF. If the test at a node cannot be applied because of insufficient information in the cue, the cue is sorted left or right randomly at that node. The program improves its performance as the number of learning trials increases, since each time it retrieves an incorrect response it enlarges the partial description connected with the retrieval of that response. Using this basic scheme EPAM is able to demonstrate stimulus generalization,

response generalization, and retroactive inhibition.

Learning takes place in EPAM by simple association; a stimulus is associated with a response cue in a terminal node. However, generalization techniques (the growing of the discrimination net and the use of partial descriptions) are employed which tend to minimize the amount of information that needs to be stored and which lead to humanlike verbal learning behavior. The important heuristics used in EPAM are those concerned with the implementation of the generalization techniques.

It is of interest to note that in all of the learning programs discussed, learning is accomplished either through rote memorization processes or through various generalization techniques. The implication here is that the process of generalization must be well understood in order to be able to construct really effective programs for performing complex learning tasks.

1.4 OBJECTIVES

This paper proposes to examine the following three questions as a first step toward the development of computer programs which learn heuristics: (1) what is a useful way of representing heuristics in a program?, (2) how can heuristics be modified by the program embodying them?, and (3) what implications do these representation and modification techniques have for theories of human learning?

Most heuristic programs (and in fact, all the programs discussed in section 1.3) have the heuristics "built-in"; i.e., the heuristics are an integral part of the program and even on close inspection it is difficult to decide exactly what heuristics are being used, what their effects are, and how they are related to one another. When this is the case, the entire program, in a sense, is a representation of the embodied heuristics.

The problem encountered in using this naive method of representation is the following. The heuristics are so entwined in the program that it is extremely difficult to make the program itself manipulate them. It would be desirable to have a program which during execution could monitor the use of its own heuristics; e.g., which could obtain measures of their values, modify them in an attempt to improve them, discard ones which seem of little value, and add new ones to replace the discarded ones. A program with the ability to manipulate its own heuristics could be given, as a secondary task, the job of learning what set of heuristics would provide optimal performance in its primary task. For instance, a game-playing program with this ability could learn, during the course of a game, how to play the game more intelligently by manipulating the

heuristics concerned with the strategy used in playing the game.

Psychologists have been studying the phenomenon of learning for over three-quarters of a century, with the result that many divergent theories or viewpoints have appeared. The majority of the work in this field has been done on simple learning (acquisition of motor skills, discrimination learning, memorization, etc.). Some work has been done on more complicated learning processes such as concept learning (Bruner, Goodnow, and Austin, 1956; Hunt, 1962), but little has been done on the complex processes involved in strategy learning in game-playing or problem-solving environments. Thus, it would prove beneficial if artificial intelligence techniques for representing and modifying heuristics could be applied to a psychological theory of complex human learning.

CHAPTER 2

REPRESENTATION OF HEURISTICS

2.1 INTRODUCTION

The feasibility of learning heuristics by dynamically manipulating them in a program depends heavily upon the method used to represent the heuristics.

REQUIREMENTS. To facilitate dynamic manipulation, the representation should satisfy the following requirements:

1. It should permit separation of the heuristics from the program using these heuristics.
2. It should provide for clear identification of individual heuristics and show how these heuristics are interrelated.
3. It should be relatively easy to work with.

The first requirement is basic, since the program would have a difficult time trying to manipulate heuristics that it could not even locate. The second requirement is necessary because individual heuristics need to be modified and evaluated, and when a modification occurs the effect of this change on the whole system of heuristics must be known if an accurate evaluation is to be made. For example, if heuristic h1 depends in some way on heuristic h2 , and h2 is modified, then effectively h1 is also modified. In the evaluation of this modification it is necessary to recognize the relation between h1 and h2 , since

it is possible that either h1 or h2 will be rendered less effective by the change. If the relation were unrecognized, the program might naively proceed with the evaluation by testing the new h2 but ignoring the heuristic h1 .

The last requirement states that the representation technique employed should be easy to work with. By this is meant (a) that the heuristics should be easy to modify or replace, (b) that the representation should be compatible with generalization schemes, and (c) that it should be easy to use the heuristics to obtain a decision from the system. The desirability of conditions (a) and (c) is clear. Condition (b) is desirable in view of the evidence presented in Chapter 1 that complex learning can be achieved through the use of generalization techniques.

The representation method discussed in Chapter 1, where the entire program is a large complex representation of the embodied heuristics, is obviously inadequate. It fails to satisfy every requirement except conditions (b) and (c) under requirement 3. This chapter will be devoted to the exposition of a representation technique which does satisfy the above requirements.

DEFINITIONS. A method of representing heuristics which satisfies the requirements of section 2.1 will now be proposed. First, however, the following items must be defined:

1. Heuristic Rule: a heuristic which directly specifies an action to be taken.

2. Heuristic Definition: a heuristic which does not specify an action directly, but instead defines a term.

3. General Heuristic: a heuristic rule or definition which employs terms defined by heuristic definitions.

4. Special Heuristic: a heuristic rule or definition which does not employ terms defined by heuristic definitions.

Some examples (taken from the game of checkers) to illustrate the above definitions are given below.

(a) If the piece advantage is "high" then 'make an even exchange'. (General heuristic rule).

(b) If the piece advantage is greater than 3 then 'make an even exchange'. (Special heuristic rule).

(c) A "high" piece advantage is one 5 or more greater than a "low" piece advantage. (General heuristic definition).

(d) A "high" piece advantage is one equal to or greater than 4. (Special heuristic definition).

In section 1.2 a heuristic is defined as a particular type of computational rule, capable of obtaining solutions to problems. Consider example (b) above from the game of checkers. This can be thought of as a computational rule for solving the problem "what type of move should I make to increase my chances of winning the game?" Furthermore, example (d) can be thought of or restated as a computational rule for solving the problem "Is the piece advantage in the present board configuration a high

one?" Thus the above definitions correspond to those presented in section 1.2.

2.2 PRODUCTION RULES

During execution, a program goes through a succession of states as the values of its variables are changed. Consider a "situation" as the set of current values of the variables of the program and let this set be called the state vector $\mathcal{E}$ of the program (McCarthy, 1962, 1965). When a block of code is executed, the effect on the state vector may be described by the equation $\mathcal{E}' = f(\mathcal{E})$, where $\mathcal{E}'$ is the resulting state vector and $f(\mathcal{E})$ is a function which stands for the block of code. In the typical heuristic program the heuristics are represented by blocks of code, each block being a complicated, inflexible function of the program variables. The relation between the code and the values of the program variables is illustrated below for variables A, B, and C with values a_1, b_1, and c_1.

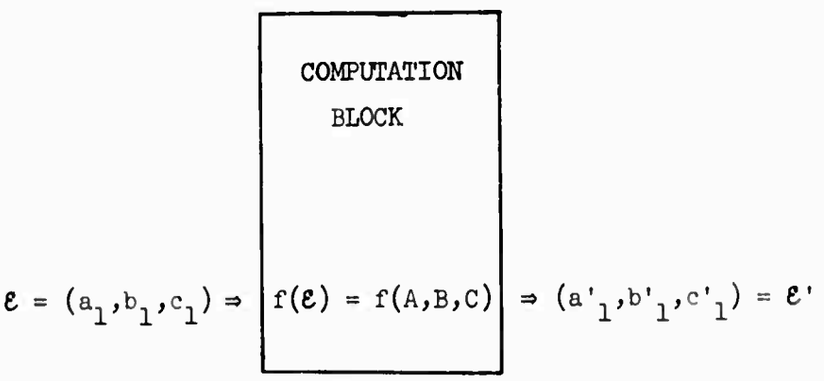


Figure 2-1.

A simple, more flexible way to express such a function is by a set of rules, each having the form

$$(a_1,b_1,c_1) \rightarrow (f_1(\mathcal{E}),\ f_2(\mathcal{E}),\ f_3(\mathcal{E}))\ .$$

The above rule states that when the value of A is a_1, B is b_1 and C is c_1, the function (or block of code) changes the values such that the value of A becomes $f_1(\mathcal{E})$, B becomes $f_2(\mathcal{E})$, and C becomes $f_3(\mathcal{E})$. The problem with this technique is that it may require an excessively large number of rules to adequately describe a function.

This difficulty can be eliminated by using sets of values in place of individual values in the description of the state vector. For example, instead of using (a_1,b_1,c_1) above to represent a particular state, (A1,B1,C1) can be used where A1, B1, and C1 are sets, in this case defined as $A1 = \{a_1\}$, $B1 = \{b_1\}$, and $C1 = \{c_1\}$. A single description such as (A1,B1,C1) can be made to represent a number of states by merely enlarging the sets defined by A1, B1, and C1. Thus by using rules of the form

$$(A1,\ B1,\ C1) \rightarrow (f_1(\mathcal{E}),\ f_2(\mathcal{E}),\ f_3(\mathcal{E}))$$

it takes fewer rules to adequately describe a function depicting a block of code containing heuristics.

In view of these considerations a heuristic will be represented as a rule of the form $\phi \rightarrow \Psi$. This rule will either (a) specify an action to be taken in situation S by the rule $S \rightarrow S'$, where S' is the situation that results after the action is taken, or (b) define a term by the rule $Z \rightarrow Z'$, where Z is the term being defined and Z' is some conbination of terms which constitutes the definition of Z.

It will be useful to think of these rules as production rules which specify how a value or string of values of variables from the state vector can lead to other strings.

REPRESENTATION OF HEURISTIC RULES. A heuristic rule can now be represented by a production rule of the type S - S' . Here S is a situation defined by the state vector variables, such as the vector (A1, B1, C1) , and S' is the definition of the resulting situation or state vector, such as $(f_1(\mathcal{E}), f_2(\mathcal{E}), f_3(\mathcal{E}))$. Production rules of the type $S \rightarrow S'$ will be called action rules (ac rules). Consequently, an action rule states that in a situation of type S the values of some of the state vector variables are changed to produce a situation of type S' . This type of production rule is weakly analogous to the productions used in a Chomsky type 0 grammar (Chomsky, 1959).

REPRESENTATION OF HEURISTIC DEFINITIONS. A heuristic definition can be represented by a production rule of the type $Z \rightarrow Z'$, where Z is a value of a state vector variable (such as A1) and Z' is either

(1) a value of a state vector variable and an associated predicate, or

(2) a computational rule for combining variables of the state vector.

Case (1) will be called a bf rule (backward form) and case (2) an ff rule (forward form). An example of case (1) is $A1 \rightarrow A$, $A > 20$, meaning that A is considered a member of the set A1 if the current value of A is greater than 20 . An example of case (2) is $X \rightarrow K1 \times D$, meaning that X is defined by the arithmetic expression K1 x D . This type of production rule is weakly analogous to the productions used in a Chomsky type 2 grammar (Chomsky, 1959).

STATE VECTOR COMPOSITION. The state vector is subdivided into three types of variables: bookkeeping variables, which provide a record of past experiences; function variables, which represent arithmetic expressions containing state vector variables; and dynamic variables, which either directly influence the decisions of the program or change in value as a direct result of these decisions. Only the dynamic variables are used in the descriptions which represent the left and right parts of the action rules.

Decision Making Using Production Rules

The production rule just described can be used to implement decision making in a problem solving program. This technique will now be illustrated for the class of problem solving programs categorized as game players. The "intelligence" of a game playing program is measured by the appropriateness of the decisions (or moves) it makes during the course of a game. In order to make a decision, a program using the production rule method of heuristic representation (1) examines the action rules to find one applicable to the current situation, and (2) uses the rule just found to change the values of certain dynamic variables of the state vector in such a way that the change defines a move.

To illustrate the use of these production rules in a game-playing situation, let the subvector β, composed of the pertinent dynamic variables of the state vector, be the following:

$$\beta = (a, b, c)$$

where A, B, and C are variables with the current values a, b, and c respectively. The heuristics to be used for this simple example are:

1. If A is an "A1" then add X to the value of B.

2. If A is an "A2" and C is a "C1" then subtract Y from the value of C .
3. If B is a "B1" then add Y to the value of C .
4. A is an "A1" when $A \geq 25$.
5. A is an "A2" when $A < 25$.
6. B is a "B1" when $B > 1$.
7. B is a "B2" when $B > 4$.
8. C is a "C1" when $C = 5$.
9. X increases as D increases.
10. Y increases as E decreases.

In the preceding heuristics, D and E are bookkeeping variables, X and Y function variables, and A, B, and C dynamic variables. The corresponding production rules are:

1.	(A1, *, *) →	(a, X+b, c)	ac
2.	(A2, *, C1) →	(a, b, c-Y)	ac
3.	(*, B1, *) →	(a, b, Y+c)	ac
4.	A1 →	A, $A \geq 25$	bf
5.	A2 →	A, $A < 25$	bf
6.	B1 →	B, $B > 1$	bf
7.	B2 →	B, $B > 4$	bf
8.	C1 →	C, $C = 5$	bf
9.	X →	K1 x D	ff
10.	Y →	K2 − (K3 x E)	ff

A "*" in a subvector indicates that the variable in question may take on any value. Hence (A1, *, *) describes all situations where A has the symbolic value A1 , while B and C have any values. Also needed are the following production rules (one for each element of the subvector):

11. A → a, a ∈ {set of possible values of A } bf

12. B → b, b ∈ {set of possible values of B } bf

13. C → c, c ∈ {set of possible values of C } bf

For this example, the set of possible values for A, B, and C will be defined as the set of natural numbers.

In the game, when the point is reached where the program must make a "move" decision, the values of A, B, C, D and E will have been set by either a previous program decision or by the non-heuristic part of the program. The terms K1, K2, and K3 are considered to be constants. The decision is made in two steps as follows.

A. Each element of the current program subvector is matched against all right sides of the bf rules. When a match occurs (the predicate is satisfied) the corresponding left side of that bf rule is then matched against all right sides of bf rules, etc., until no more matches can be found. The resulting set of symbols defines a symbolic subvector. This step is somewhat analogous to parsing (Irons, 1964; Ingerman, 1966).

B. The symbolic subvector derived in Step A is matched against all left sides of the action rules, going from top to bottom, and when the first match is found the values of the program subvector are modified as described by the right side of the matched rule. A forward search is usually necessary, through the ff rules, to determine the new values for the program subvector variables.

As a concrete example let the subvector have the values a = 4, b = 5, c = 6 , the constants have the values K1 = 1, K2 = 20, K3 = 3 , and let the bookkeeping variables have the values D = 7 and E = 8 . Then β = (4, 5, 6) and the "parse" of step A has the following form.

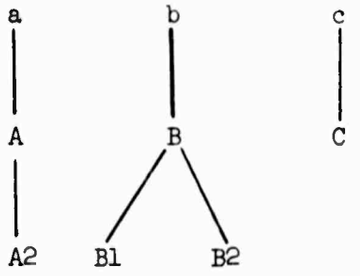


Figure 2-2.

Here step A is initiated by comparing a = 4 with each bf rule predicate, the predicate being satisfied only if it contains the symbol a and is true when a is set equal to 4 . Thus a = 4 is found to match rule 11 and no others. Next, A = 4 is similarly compared with all bf rule predicates and is found to match only rule 5. Finally, A2 = 4 is compared with all bf rule predicates, and since it matches none of them the search terminates, leaving A2 as the final symbolic value. Elements b and c are processed in the same manner, and the symbolic subvector that results is ((A2), (B1,B2), (C)) . This subvector is a description of all situations in which (1) the variable A has the symbolic value A2 , (2) the variable B has either the symbolic value B1 or B2 , and (3) the variable C has the symbolic value C .

Step B now consists of comparing the subvector ((A2), (B1,B2), (C)) with the left side of each action rule, until a match is found. In this case a match occurs at rule 3. The program subvector is then set

to the values specified in the right side of rule 3. Hence the new β equals $(4, 5, (20 - (3 \times 8)) + 6)$ or $(4, 5, 2)$. In effect, the program made the decision to change the value of the variable C to 2.

The method just proposed for representing heuristics easily satisfies the first two requirements of section 2.1, since the heuristics are separated from the program, and the individual heuristics and their interrelationships are clearly identified. The third requirement of section 2.1 is also satisfied, since the production rules are easy to modify or replace, are compatible with generalization schemes (this will be shown in Chapter 3), and are easy to use to obtain a decision from the system. Standard techniques for handling production rules, such as parsing, are seen to suggest methods which can be used to facilitate the decision making process.

NEWELL'S SYSTEM. This is not the first attempt to use a production system as the underlying mechanism in a problem solving scheme. Newell (1966, 1967) uses a production system to characterize the problem solving process occuring in a human subject as he solves crypt-arithmetic problems. Each production consists of an expression of the form:

condition → action

and specifies the action to take when the condition in the left part of the production is true. The productions are priority ordered so that the system can uniquely determine which production to use in situations where more than one is applicable. The production rule system just described closely parallels Newell's system in its general approach to decision making.

2.3 TRANSLATION OF HEURISTICS INTO PRODUCTION RULES

At this point it is reasonable to ask how one can go from a heuristic stated informally, like "if the piece advantage is high make an even exchange", to a set of representative production rules. This transition can be accomplished through the use of an intermediate step, that is, a formal language in which heuristics can be expressed precisely, and which can be automatically translated into production rules. With such a tool, one would only have to restate the heuristic in this intermediate formal language in order to effect its transformation into production rules.

A Language For Specifying Heuristics

The syntax of a language for expressing heuristics is presented in Figure 2-3 as a set of syntactic rules. This language will be called LASH: language for specifying heuristics.

TERMINAL SYMBOLS. The terminal symbols in the syntactic rules include (1) all the underlined words, (2) all non-alphabetic symbols, and (3) all Greek letters. The terminal symbol @ stands for any ALGOL-like identifier (Bauman et al., 1964; Ekman and Froberg, 1965), while the terminal symbol # stands for any ALGOL-like number.

The terminal symbol λ stands for any simple arithmetic expression, that is, any ALGOL-like expression composed of identifiers, the arithmetic operators +, -, x, ÷ and the delimiters) and (. However one restriction is made; a single number or identifier must be enclosed in parentheses to be recognized as an expression. Without this restriction it would be, in some cases, impossible to determine whether a given

terminal string was an @ , a # , or a λ . Also, one extension is made; an expression can include the function "random (a,b)", which when executed evaluates to a number chosen at random from the range a to b .

The terminal symbol π stands for any simple Boolean expression which is enclosed in parentheses, that is, any parenthesized ALGOL-like Boolean expression composed of identifiers, arithmetic operators +, -, x, ÷, relational operators > , < , = , ≠ , and the delimiters) and (. Some examples of @-type strings are K1, STORE, and M3J , of #-type strings are 3, 1.5, and -12 , of λ-type strings are (K1), (3), and L8 + (3 x Q) , and of π-type strings are (P > 4), (6 x M4 = PL-3), and (L8 + (3 x Q) < K1) .

routine ⟶ begin declarationset . routinel end

routinel ⟶ rules . definitions

routinel ⟶ rules

declarationset ⟶ declarations

declarations ⟶ declarations , declaration

declarations ⟶ declaration

declaration ⟶ actionname : action

rules ⟶ rules otherwise rule

rules ⟶ rule

rule ⟶ if predicate then statement

statement ⟶ (rule else statement)

statement ⟶ actionname

predicate ⟶ pred

pred ⟶ pred1 ∧ pred

pred ⟶ pred1

pred1 ⟶ pred1 ∨ pred2

pred1 ⟶ pred2

pred2 ⟶ atom relation number

pred2 ⟶ atom relation1 atom

actionname ⟶ 'atom'

action ⟶ assign ; action

action ⟶ assign

assign ⟶ atom ⟵ expression

definitions ⟶ definition , definitions

definitions ⟶ definition

definition ⟶ atom equals expression

definition ⟶ atom is art atom such that fullpred

art ⟶ a

art ⟶ an

relation ⟶ =

relation ⟶ ≠

relation ⟶ <

relation ⟶ >

relation ⟶ ≤

relation ⟶ ≥

relation1 ⟶ ≡

atom ⟶ @

number ⟶ #

expression ⟶ λ

fullpred ⟶ π

Figure 2-3. Syntax of a language for specifying heuristics.

SIMPLE PRECEDENCE SYNTAX. The syntax presented in Figure 2-3 is a simple precedence syntax i.e., the syntactic rules are so arranged that the relation between any two symbols is unique. Three types of relations are considered.

(1) The relation $\doteq$ holds between all adjacent symbols within any string forming the right side of a syntactic rule.

(2) The relation $\lessdot$ holds between the symbol immediately preceding a reducible string and the leftmost symbol of that string.

(3) The relation $\gtrdot$ holds between the rightmost symbol of a reducible string and the symbol immediately following that string.

Here a reducible string is one which can be reduced through parsing to another string of equal or smaller length. As a consequence of this arrangement, the language defined by the syntax is a simple precedence phrase structure language (Wirth and Weber, 1966).

The advantage in using this type of language is that there exists a very efficient algorithm for parsing sentences of the language (Wirth and Weber, 1966). This is quite important if one wants to construct a syntax-directed compiler (Irons, 1961, 1963; Ingerman, 1966) for automatically translating the language into some other form, such as a set of machine instructions or list of rules. Thus the language is designed not only to provide for adequate descriptions of heuristics, but also to permit relatively simple and efficient translation into production rules. The computer program to be described in this paper does not include a compiler for translating LASH into production rules. Consequently, translation into production rules is performed by hand.

STRUCTURE. The structure of the language defined in Figure 2-3 will now be illustrated by using it to express a number of heuristics for a hypothetical game. It will be assumed that for this game the dynamic variables are A, B, C, D, and E , the bookkeeping variables are F and G , the function variables are P and R , and the constants are K1, K2, K3, and K4 . The way in which the language can be used to express heuristics is shown below.

```
begin 'MOVE1' : B ← 2xB; C ← D +(4xC)+P,
      'MOVE2' : B ← B+6; D ← C+D; E ← (0),
      'MOVE3' : A ← (5); D ← (E).
if A > 5 ∧ B < 10 then 'MOVE1' otherwise
if A > 20 then (if B=0 then 'MOVE2' else
                (if B=1 ∧ C≡CX then 'MOVE3' else 'MOVE1')) otherwise
if D≡DZ then 'MOVE3' .
      CX is a C such that (C+5 > P),
      DZ is a D such that (D < E-20),
      P equals (K1 x F) - (K2 x R),
      R equals (K3 x G) + (K4 x A)   end
```

Note that each of the three declarations, MOVE1, MOVE2, and MOVE3 , define a change to be made in the state vector, or more precisely a change in some of the dynamic variables of the state vector. The three rules (see Figure 2-3 for the definition of the symbol "rule") in the above example specify under what conditions each of these changes in the state vector is to be made. The four definitions contained in the example merely define variables used in the declarations, the rules and in the definitions themselves.

TRANSLATION. The heuristics in the above example translate into the following production rules.

(A1, B1, *, *, *)	→	(*, 2xb, d+(4xc)+P, *, *)	ac
A1	→	A, A > 5	bf
B1	→	B, B < 10	bf
(A2, B2, *, *, *)	→	(*, b+6, *, c+d, 0)	ac
(A2, B3, CX, *, *)	→	(5, *, *, e, *)	ac
(A2, *, *, *, *)	→	(*, 2xb, d+(4xc)+P, *, *)	ac
A2	→	A, A > 20	bf
B2	→	B, B=0	bf
B3	→	B, B=1	bf
(*, *, *, DZ, *)	→	(5, *, *, e, *)	ac
CX	→	C, C+5 > P	bf
DZ	→	D, D < e-20	bf
P	→	(K1xF) - (K2xR)	ff
R	→	(K3xG) + (K4xA)	ff

Here when the value of a variable in the right side of an action rule is a "*" it means that no change is made in the value of that variable. Thus

(A2, B3, CX, *, *) → (5, *, *, e, *)

means that when A≡A2 , B≡B3 , and C≡CX then A is changed to 5 , D is changed to the current value of E , and B, C, and E are left unchanged in value. This notation is slightly different from (and slightly superior to) the notation presented earlier for the representation of heuristic rules. In the earlier notation the above rule would be

written

$$(A2, B3, CX, *, *) \rightarrow (5, b, c, e, e) .$$

It should be noted that a rule in LASH translates almost directly into a number of action rules and bf-type heuristic definitions. Moreover, a definition in LASH translates directly into either an ff-type or a bf-type heuristic definition. Thus the translation of heuristics expressed in this language into production rules is a relatively simple task.

SPECIFYING HEURISTICS IN LASH. There is one question as yet unanswered. How difficult is it to take heuristics stated in natural language and restate them in this formal language? The answer is that it is quite easy to make this transition, provided that a relevant state vector has been established and its variables defined. For example, the heuristic mentioned at the beginning of this section, "if the piece advantage is high make an even exchange", can be restated as

if PIECEADVANTAGE ≡ HIGH then 'EVENEXCHANGE' .

Also necessary is (1) a LASH declaration defining 'EVENEXCHANGE' by specifying the effect of an even exchange on the state vector variables, and (2) a LASH definition defining the term HIGH. The high degree of similarity between the heuristic stated in English and the heuristic stated in LASH indicates how simple, sometimes even trivial, the transition from one to the other can be. Consequently the formal language serves as a very convenient intermediate step in the process of translating heuristics into production rules.

CHAPTER 3

PROGRAM MANIPULATION OF HEURISTICS

3.1 CREATION AND EVALUATION OF HEURISTICS

Ideally, a heuristic problem-solving-program should be able to modify or replace its heuristics in order to improve its overall problem solving performance. A step has been made in this direction by the development of a game playing program which modifies coefficients in an evaluation polynomial in order to improve performance (Samuel, 1959, 1960), and a pattern recognition program which generates, evaluates, and modifies its operators in an attempt to improve pattern recognition ability (Uhr and Vossler, 1961). However, these programs make no effort to recognize, create or evaluate individual heuristics, and as a consequence they are unable to radically modify their own heuristic configurations.

Before the manipulation of heuristics in a program can be implemented two major problems must be faced:

(1) the problem of evaluating existing heuristics in terms of their usefulness to the program.

(2) the problem of creating new heuristics, both by modifying old ones and hypothesizing new ones.

To solve these problems, techniques must be devised which will enable the program to evaluate and create heuristics during the course of its regular problem solving activity.

Evaluation of Heuristics

Of the two problems just outlined, the first one, measuring the value

or usefulness of a heuristic is perhaps the more difficult. This problem is actually an excellent example of the basic credit-assignment problem for complex reinforcement learning systems (Minsky, 1961).

CREDIT-ASSIGNMENT PROBLEM. The credit-assignment problem is the following. If a large number of steps are required to complete some complex task, then how should the credit for completing the task be distributed among each of the individual steps? A learning system which could answer this question would be able to reinforce steps pertinent to completion of the task and thus learn which steps are necessary and which are redundant or ineffectual. A rudimentary solution to the credit-assignment problem is to merely assign an equal amount of credit to each step involved in the successful completion of the task. This approach, however, will lead either to very inefficient learning or no learning at all unless the steps are relatively independent. If the steps are highly dependent, as is the case for the tasks to be considered in this paper, this simple approach is doomed to failure.

Minsky (1961) illustrates the dangers of underrating the credit-assignment problem in a discussion of a program-writing program by Friedberg (1958, 1959). The Friedberg program is designed to learn, through reinforcement, to write a test program that will perform some simple task. Friedberg's program attempts this by (a) randomly generating a 64-instruction test program, (b) executing this test program and evaluating its operation according to a predetermined criterion, and (c) using the information concerning the success or failure of the test program to reinforce individual instructions associated with successful test programs. Reinforcement consists of increasing the probability that particular instructions will be generated in later trials. Friedberg's program

learns to solve simple problems but takes much longer than it would take to solve the problems by pure chance alone. The mistake made, Minsky notes, is that credit is assigned to individual instructions rather than to functional groups of instructions such as subroutines, and this disregard for the hierarchical nature of the problem leads to the poor results.

OUTER-LEVEL PROBLEM. Evaluating or measuring the usefulness of a heuristic in a game playing program (or any type of problem solving program) is actually a 2-level credit-assignment problem; that is, a credit-assignment problem within another credit-assignment problem. The outer or top-level problem is to evaluate the effectiveness of a sequence of decisions or "moves" and then to use this result to assign credit or blame to the individual decisions in the sequence. The problem is difficult because it may not be clear how to distribute the credit or blame. For example, if the sequence is a poor one, which decisions in the sequence should take the blame? It would be unrealistic to blame every decision automatically, since the sequence may have been ruined by just one or two key decisions. Conversely, if the sequence is a good one it does not necessarily mean that every decision is good; there could be a few poor ones present which exert very little influence on the game situation.

In general, it is relatively easy to evaluate the effectiveness of a long sequence of game decisions (the longer the sequence, the easier the evaluation) but difficult to evaluate or determine the effectiveness of any individual decision. Even so, it must be pointed out that the method used to determine the value of a game decision depends to a large

extent on the particular game under consideration.

INNER-LEVEL PROBLEM. The inner or lower-level credit-assignment problem is that of using the evaluation of a game decision to assign credit or blame to the individual heuristics which played a part in making the decision. Again the problem is difficult because there exists no simple rule for specifying how to distribute the credit or blame. This problem is possibly more formidable than the higher-level problem, since the heuristics are often highly entangled and interdependent. Assigning credit (or blame) to a set of heuristics which have been involved in making a good (or bad) decision entails trying to determine to what degree each heuristic contributed to the decision. This is especially difficult when the heuristics are very dependent on one another.

SOLUTION TO THE EVALUATION PROBLEM. Part of the solution to the problem of evaluating heuristics lies in the method chosen to represent them. The first step in solving the problem is obviously to separate the heuristics from the main body of the program and to clearly define the relationships existing between them. This is accomplished automatically by representing heuristics as production rules. The next step is to devise techniques for distributing credit or blame. The heirarchical arrangement of the production rules in the form of an ordered list suggests the following type of analysis. When a decision is made via production rules a symbolic subvector representing the game situation is compared to all left parts of the list of action rules (production rules which represent heuristic rules) going from top to bottom until a match is found. The action rule which defines the decision, that is, the one

whose left part matches the symbolic subvector, can easily be located. After the decision is evaluated the credit or blame can then be assigned to the action rule which defined the decision (or to the rules above it in the list of action rules) and to the associated heuristic definitions. The approach to be used here is that of assigning blame to action rules leading to poor decisions by immediately modifying these rules in an attempt to make them more effective, while ignoring action rules leading to good or acceptable decisions.

Creation of Heuristics

The second major problem which must be faced before the heuristics of a program can be adequately manipulated is the problem of creating new heuristics. The most feasible way of creating new heuristics is by modifying existing ones. For action rules, three modification techniques will be considered:

(1) Replacing the symbolic values in the left part of the rule. For example, (A1, B1, *) → (1, 2, *) might be changed into (A, B3, *) → (1, 2, *) .

(2) Changing the relevancy of the elements in the left part of the rule. For example, (A1, B1, *) → (1, 2, *) might be changed into (*, B1, *) → (1, 2, *) . Here element A is made irrelevant.

(3) Changing the heuristic definitions associated with the left part of the rule. For example, (A1, B1, *) → (1, 2, *) might remain unaltered while the definition of A1 is changed; i.e., A1 → A , A < 15 might become A1 → A, A < 20.

These techniques will be applied to action rules which lead to

decisions that are evaluated as being poor. Heuristic definitions represented by bf-type rules will be modified by simply changing the predicates in the right parts of the rules. Definitions represented by ff-type rules will not be modified.

INFORMATION NEEDED. In order to create useful heuristics, either by modifying existing ones or by hypothesizing new ones, three items of information will be used.

(1) a good or acceptable decision for the situation,

(2) the situation elements (subvector variables) relevant to making this good decision, and

(3) the reason why the decision is being made, expressed as an evaluation of these relevant situation elements.

To illustrate that these three items are adequate consider the example given below. The subvector β for this example will be defined by the dynamic variables A, B, and C . The action rules will be

1. (A1, *, C2) → (*, *, c+3)
2. (A2, B1, *) → (a+2, *, *)
3. (*, B2, C1) → (*, b+1, *)

and the rules corresponding to heuristic definitions will be

4. A1 → A, $A \geq 20$
5. A2 → A, $A < 20$
6. B1 → B, $B \geq 16$
7. B2 → B, $B < 16$
8. C1 → C, $C > 5$
9. C2 → C, $C \leq 5$
10. A → a, $a \in$ {set of natural numbers}

11. B → b ∈ {set of natural numbers}

12. C → c ∈ {set of natural numbers}

If the program subvector representing the game situation is considered to be (13, 5, 7) , the symbolic subvector obtained through parsing is (A2, B2, C1) . This symbolic subvector matches rule 3 above and leads to the decision of incrementing the value of B by 1 . If it can be determined that this was a poor decision and that

(1) a good decision is to add 6 to the value of A ,

(2) the variables relevant to this decision are A and C , and

(3) the decision is being made because the current value of A classifies A as an A1 and the current value of C classifies C as a C1 ,

then the production rules can be modified by (a) changing the rules corresponding to the heuristic definitions of A1 and A2 such that they become A1 → A , A ≥ 13 and A2 → A , A < 13 , and (b) inserting the action rule (A1, *, C1) → (a+6, *, *) just above the action rule which now "catches" the symbolic subvector. Changing the definitions of A1 and A2 changes the symbolic subvector to (A1, B2, C1) which still matches or catches on rule 3, thus the new action rule is inserted just above rule 3. After such a modification is made the rules have the form:

1. (A1, *, C2) → (*, *, c+3)

2. (A2, B1, *) → (a+2, *, *)

3. (A1, *, C1) → (a+6, *, *)

4. (*, B2, C1) → (*, b+1, *)

5. A1 → A, $A \geq 13$

6. A2 → A, $A < 13$

7. B1 → B, $B \geq 16$

8. B2 → B, $B < 16$

9. C1 → C, $C > 5$

10. C2 → C, $C \leq 5$

11. A → a, a ∈ {set of natural numbers}

12. B → b, b ∈ {set of natural numbers}

13. C → c, c ∈ {set of natural numbers}

It can be seen that now in the situation (13, 5, 7) the correct decision, "add 6 to the value of A ", is made. Consequently, the three items of information previously mentioned, i.e., a good decision, the relevant elements, and an evaluation of these elements, permit the creation of useful or "good" heuristics. This process is specified in detail in the next section.

3.2 TRAINING PROCEDURES

In the previous section it was noted that three items of information are adequate for the creation of useful heuristics:

(1) a good decision for the situation,

(2) the relevant situation elements, and

(3) the reason why the decision is being made.

When a learning program is presented with a game situation and the above items of information for the purpose of improving its performance, the process will be called training.

BOOK LEARNING. In section 1.3 a checker-playing program which employs an abbreviated form of training is described. This technique is called book learning (Samuel, 1959, 1967), a procedure wherein the program is presented with game situations and the associated book-recommended moves and is permitted to use this book information to correct its move-generating apparatus. In this procedure item (1) above is given to the program but items (2) and (3) are not.

Book learning has proved to be a successful technique for teaching programs to play games where minimaxing procedures can be applied. The book information supplies the program with a good move decision while the minimaxing procedure provides a method by which the program can determine which situation elements (or parameters) are relevant. One way parameter relevancy is determined in the checker program is by comparison of the current parameter values for a situation with the backed-up parameter values obtained through minimaxing on the path in the game tree corresponding to the book move. The parameters whose backed-up values are

consistently greater than the current values are considered the relevant ones, since these are the parameters that the book moves tend to increase. In one version of the checker program the value or worth of any game situation (or board configuration) is represented by a linear polynomial. As a consequence, when a move decision is made it is always because the move has associated with it the largest numerical value obtained by minimaxing evaluations of the polynomial back up the game tree. Thus by using minimaxing and a polynomial representation of the board value the program is able to obtain, by itself, the information specified by items (2) and (3) above.

TRAINING. For the general game-playing program, where the parameters are not independent and minimaxing is impossible (because not enough information is known to construct a game or decision tree) training procedures can be used to improve performance. This training can take place in two ways, (a) by supplying the program with a number of unrelated game situations and the associated information needed for training, or (b) by having a human (who is an expert at the game) monitor the decisions of the program as it plays an actual game and give the program, when a poor decision is made, the three items of training information.

In section 3.1 an example was presented which indicated how heuristics in production rule form can be created or learned when the appropriate training information is available. The use of training information in learning heuristic rules and definitions will now be examined in detail.

Learning Heuristic Rules

As illustrated in section 3.1 the training information provides the data necessary for the construction of a new action rule; i.e., item (1)

of the training information supplies the right part of the action rule, while items (2) and (3) supply the left part. The most elementary method of correcting the set of action rules when they lead to a poor decision is by (a) using the training information to create a new action rule through generalization, and (b) inserting this new rule in the list of action rules immediately above the action rule which led to the unacceptable decision. However, this method may not always be practical, since it entails adding a new action rule for every training trial. Such a technique could lead to a prohibitive number of action rules.

CORRECTION BY MODIFYING EXISTING RULES. What is needed for efficient correction of the set of action rules is the addition of another generalization scheme to the abovementioned process. Such a scheme should permit training information to be added to the set of action rules without the insertion of a new rule. One way this can be accomplished is by finding an appropriate action rule already located above the error-causing rule and modifying it to make it general enough to catch the symbolic subvector. An appropriate rule is one which is capable of being suitably modified and which leads to the same decision as that specified in item (1) of the training information. After such a modification is carried out, the training information is effectively incorporated into the set of action rules. This is true because whenever the original training situation is re-encountered (i.e., the current state vector is identical to the state vector of the training trial) the system will make the decision previously specified by the training information.

If no appropriate rules are located above the error-causing

rule but some are located below it, the following approach may be used. The error-causing rule, if suitable, is modified so as to pass (rather than catch) the symbolic subvector, while the first appropriate action rule below it is modified to catch the subvector. Also, if any rules located between the error-causing one and the first appropriate one catch the subvector, they are modified to pass it. This type of modification also incorporates the training information into the set of action rules.

RULES APPROPRIATE FOR MODIFICATION. At this point it must be made clear which rules can be modified to catch the symbolic subvector, which can be modified to pass it, and exactly how this modification process takes place. An action rule will be considered appropriate for modification to catch the subvector if it has the same form as the training rule, that is, the action rule which can be created from the training information. An action rule has the same form as the training rule only if (1) their right parts are identical, (2) for each * in the left part of the training rule there is a corresponding * in the left part of the action rule, and (3) the corresponding symbolic values of their left parts are identical, or at least are alike to the extent that they are both defined by the same logical operator. Here * is considered to always be identical to any other symbolic value.

EXAMPLE OF RULE MODIFICATION. For example, consider the rule created from the training information to be

$$(A1, *, C1) \rightarrow (*, b+2, *)$$

and the existing production rules to be

1. (A1, *, C2) → (*, b+2, *)
2. (A1, B1, *) → (*, *, a+5)
3. (A2, *, C3) → (*, b+2, *)
4. (A1, *, *) → (*, *, a+5)
5. A1 → A, A < 6
6. A2 → A, A < 8
7. B1 → B, B > 8
8. C1 → C, C > 12
9. C2 → C, C < 5
10. C3 → C, C > 14

Here rule 1 and the training rule are not of the same form because C1 and C2 are not defined by the same logical operator (requirement (3) above). Rule 2 and the training rule are not of the same form because rule 2 has a B1 where the training rule has a * and their right parts are different (requirements (2) and (1) above). Rule 3 and the training rule, however, are of the same form since they satisfy all three of the above requirements.

An action rule can be modified to catch the symbolic subvector by enlarging the sets defined by the symbolic values in the rule. As an illustration of this generalization technique consider again the example just presented, and let the program subvector be (5, 3, 13) . The symbolic subvector obtained through parsing is ((A1, A2), (B), (C1)), which matches or catches on rule 4. This rule leads to a poor decision, since it is not the decision advocated by the training information. Rule 3 is located above error-causing rule 4 and has the same form as the training rule. Thus, if rule 3 is modified to catch the symbolic subvector, the training rule will effectively be incorporated into the

set of action rules. The left part of rule 3 is (A2, *, C3) , so it can be seen that the subvector matches the left part of rule 3 with respect to its first two elements but not with respect to its third element C3 . If the value C3 in rule 3 is replaced by a symbolic value representing a set large enough to include the current value of the state vector variable C (which in this case is 13) the symbolic subvector obtained through parsing will catch on rule 3. Therefore C3 is replaced by C1 , making rule 3 become (A2, *, C1) → (*, b+2, *) . The subvector now catches on rule 3, as desired, and causes the action advocated by the training information to be taken.

An action rule can be modified to pass the symbolic subvector by reducing the size of the sets defined by the symbolic values in the rule. This technique is somewhat the opposite of the generalization method just described. In the previous example the symbolic subvector catches on the new rule 3. To modify this rule so that it passes the subvector it is necessary to restrict the definition of one of the symbolic values in the rule such that the symbolic subvector no longer includes this symbolic value. This can be achieved by restricting the definition of A2 so that it no longer includes the current value of the state vector variable A (which in this case is 5). Let rule 6 become A2 → A, A < 5 ; then the symbolic subvector becomes ((A1), (B), (C1)) which fails to catch on the new rule 3, as desired.

OVERGENERALIZATION. When an action rule is modified so it will pass (or catch) the symbolic subvector it is necessary to expand (or restrict) the size of the sets defined by one or more of the symbolic values in the rule. Care must be taken not to overgeneralize, that is, to change

the definitions of the symbolic values. If this happens the training process could become unstable; that is, many redundant action rules might be created during training.

Overgeneralization may be guarded against by specifying the maximum allowable definition change which may be made. In the previous examples C1 replacing C3 led to a change of size 2, since the predicate was changed from $C > 14$ to $C > 12$, and A2 had a definition change of size 3. The maximum allowable change depends largely on the type of game being played, and thus will be represented as a generalization constant K which can be changed only by the programmer. In view of these considerations, an action rule is appropriate or suitable for modification only if the definition change involved is equal to or less than K.

Learning Heuristic Definitions

It has been shown how the three items of training information supply the data necessary for the creation and modification of heuristic rules represented as action rules. This training information also provides the necessary data for creating or learning heuristic definitions represented as bf rules. The techniques which can be used to learn heuristic definitions will now be described.

PARTITIONING. A simple bf rule consists of a production rule and an associated simple predicate, such as

$$A1 \rightarrow A,\ A > 10$$

This rule states that if the value of the state vector variable A is greater than 10, then the state vector variable A may take on the symbolic value A1. The symbolic values a state vector variable may

take partition the set of possible values for that variable into subsets. Two types of partitioning procedures will be considered, (1) mutually exclusive (and exhaustive) partitioning, and (2) overlapping (and non-exhaustive) partitioning. An example of mutually exclusive partitioning for the state vector variable A is

$$A1 \rightarrow A,\ A > 10$$

$$A2 \rightarrow A,\ A \leq 10$$

where the set being partitioned is just the set of natural numbers. Here any value of the state vector variable A permits A to take one and only one symbolic value. An example of overlapping partitioning is

$$A1 \rightarrow A,\ A > 10$$

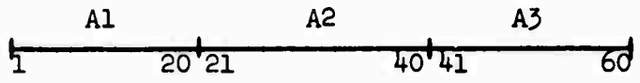
$$A2 \rightarrow A,\ A > 4$$

Here a particular value of the state vector variable A may permit A to take zero, one, or a number of symbolic values.

EXCLUSIVE VS OVERLAPPING VARIABLES. In the learning procedure about to be outlined a state vector variable will be considered one of two types: either an exclusive variable with symbolic values defined by mutually exclusive definitions, or an overlapping variable with symbolic values defined by overlapping definitions. Item 3 of the training information provides a reason why the proposed decision is being advocated. When an exclusive state vector variable is being referred to in item 3, the symbolic value associated with the current numerical value of the variable must be given. Let A , for example, be an exclusive state vector variable with a value of 8 . Then item 3 might state that the

proposed decision is being advocated because " A is an A2 ". When an overlapping state vector variable is being referred to in item 3, a magnitude indication associated with the current numerical value of the variable must be given. Let A , for example, be an overlapping state vector variable with a value of 20 . Then item 3 might state that the proposed decision is being advocated because " A is large" or because " A is small".

LEARNING EXCLUSIVE DEFINITIONS. The procedure for learning the definitions of the symbolic values of an exclusive state vector variable merely consists of partitioning the given range into the number of desired subsets and then using the data of item 3 from each training trial to shift the boundary lines whenever the newly acquired information so permits. An example will clarify this procedure. Let A be an exclusive state vector variable with the three subsets or possible symbolic values A1 , A2 , and A3 , and let the range of A be the positive integers from 1 to 60 . Initially A is partitioned into the specified number of subsets by estimating or guessing the boundary locations. Let the initial estimate of the boundaries partition A as follows:



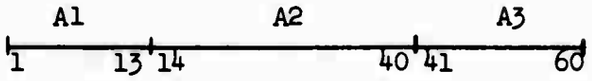
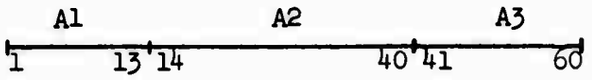
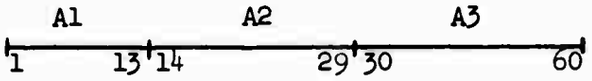
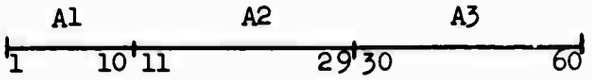
Thus the initial bf rules are

$$A1 \rightarrow A,\ A \leq 20$$

$$A2 \rightarrow A > 20 \wedge A \leq 40$$

$$A3 \rightarrow A,\ A > 40$$

The effect of 4 hypothetical training trials on the partitioning is shown below.

Trial	Information	New Boundaries
1.	A = 14, A has the value associated with the middle subset; i.e., A is an A2	
2.	A = 7 , A is an A1	
3.	A = 30 , A is an A3	
4.	A = 11, A is an A2	

The bf rules learned are:

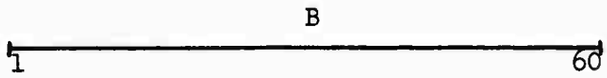
$$A1 \rightarrow A,\ A \leq 10$$

$$A2 \rightarrow A,\ A > 10 \wedge A \leq 29$$

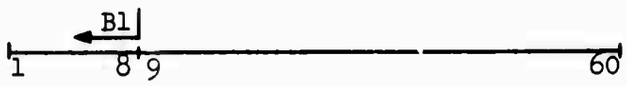
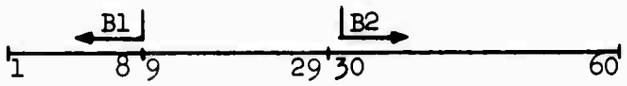
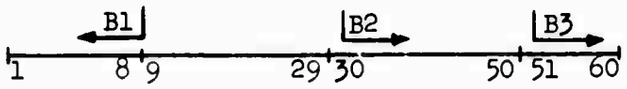
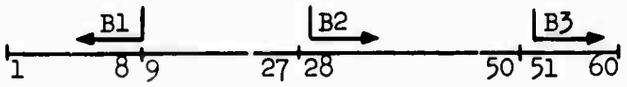
$$A3 \rightarrow A,\ A > 29$$

LEARNING OVERLAPPING DEFINITIONS. The procedure for learning the definitions of the symbolic values of an overlapping state vector variable is quite elementary. It consists of using the magnitude indication of item 3 together with the current numerical value of the state vector variable to define a particular subset of the range. If the variable is classified as "large" the current numerical value of the variable and all values above it are defined as a subset. Conversely, if the classification is "small" the current value and all below it are defined as a subset. Consider the following example for the overlapping state vector variable B

with a range from 1 to 60 . Initially, there are no bf rules for B , and the range is unpartitioned as follows:



The effect of 4 hypothetical training trials is shown below.

Trial	Information	New Boundaries
1.	B = 8 , B is small	
2.	B = 30 , B is large	
3.	B = 51 , B is large	
4.	B = 28 , B is large	

Note that on trial 4 instead of defining a new subset B4 , where B > 27 , the existing subset B2 was enlarged. This type of generalization will be performed whenever it can be accomplished without enlarging beyond some maximum amount KK , a constant which depends on the game being learned. The bf rules learned are:

$$B1 \rightarrow B,\ B < 9$$

$$B2 \rightarrow B,\ B > 27$$

$$B3 \rightarrow B,\ B > 50$$

Training Procedure Outline

The entire training procedure for learning heuristics represented as production rules will now be briefly outlined. This outline, shown

below, lists the steps involved in a single training trial.

1. a. Parse the program subvector to obtain the symbolic subvector.
   b. Drop the symbolic subvector through the action rules to obtain a decision.
   c. If the trainer indicates that the decision was acceptable then stop, otherwise go to step 2.
2. a. Obtain the training information from the trainer.
   b. Construct an action rule (to be called the training rule) from this information.
   c. Use item (3) of the training information to change or create bf rules which represent heuristic definitions. If this changes the symbolic subvector then go to step 3, otherwise go to step 4.
3. a. Drop the new symbolic subvector through the action rules to obtain a decision.
   b. If the decision is the one advocated by item (1) of the training information then stop, otherwise go to step 4.
4. a. Locate the action rule responsible for the unacceptable decision made in step 3 (or in step 1 if step 3 was skipped). This action rule will be called the error-causing rule.
5. a. Search the action rules above the error-causing rule for a rule which has the same form as the training rule and is suitable for modification to catch the symbolic subvector. This rule will be called the target rule.
   b. If such a rule is found modify it to catch the symbolic subvector and go to step 3, otherwise go to step 6.

6. a. Search the action rules below the error-causing rule for a rule which has the same form as the training rule and is suitable for modification to catch the symbolic subvector. This rule will be called the target rule.

   b. If (1) such a rule is found, (2) the error-causing rule is suitable for modification to pass the symbolic subvector, and (3) the rules between the error-causing rule and the target rule either pass the symbolic subvector or are suitable for modification to pass it then modify the target rule to catch the subvector, the error-causing rule to pass the subvector, and the rules between these two to pass the subvector and go to step 3, otherwise go to step 7.

7. a. Place the training rule immediately above the error-causing rule in the list of action rules and stop.

These steps are illustrated by the block diagram given in figure 3-1. To see exactly how these steps are applied consider the following example, where the dynamic subvector variables are A, B, and C . Here A is an exclusive variable, while B and C are overlapping variables. The initial set of production rules for this example is shown below.

1. (A2, B1, *) → (a+1, *, *)
2. (A1, *, C1) → (*, b+2, *)
3. (*, *, *) → (random)
4. A1 → A, A < 20
5. A2 → A, A ≥ 20
6. B1 → B, B > 3
7. C1 → C, C > 9

The word random in the right part of rule 3 means that if the symbolic subvector catches on this rule, a decision will be chosen at random from the set of possible decisions. During training "random" is assumed to always lead to an unacceptable decision since this accelerates the training process.

INSERTING A NEW ACTION RULE. Let the program subvector at the beginning of trial 1 be (18, 2, 11) . This parses to the symbolic subvector (A1, B, C1) which catches on rule 2 and leads to the decision of incrementing B by 2 . Assume that this decision is unacceptable and that the training information is:

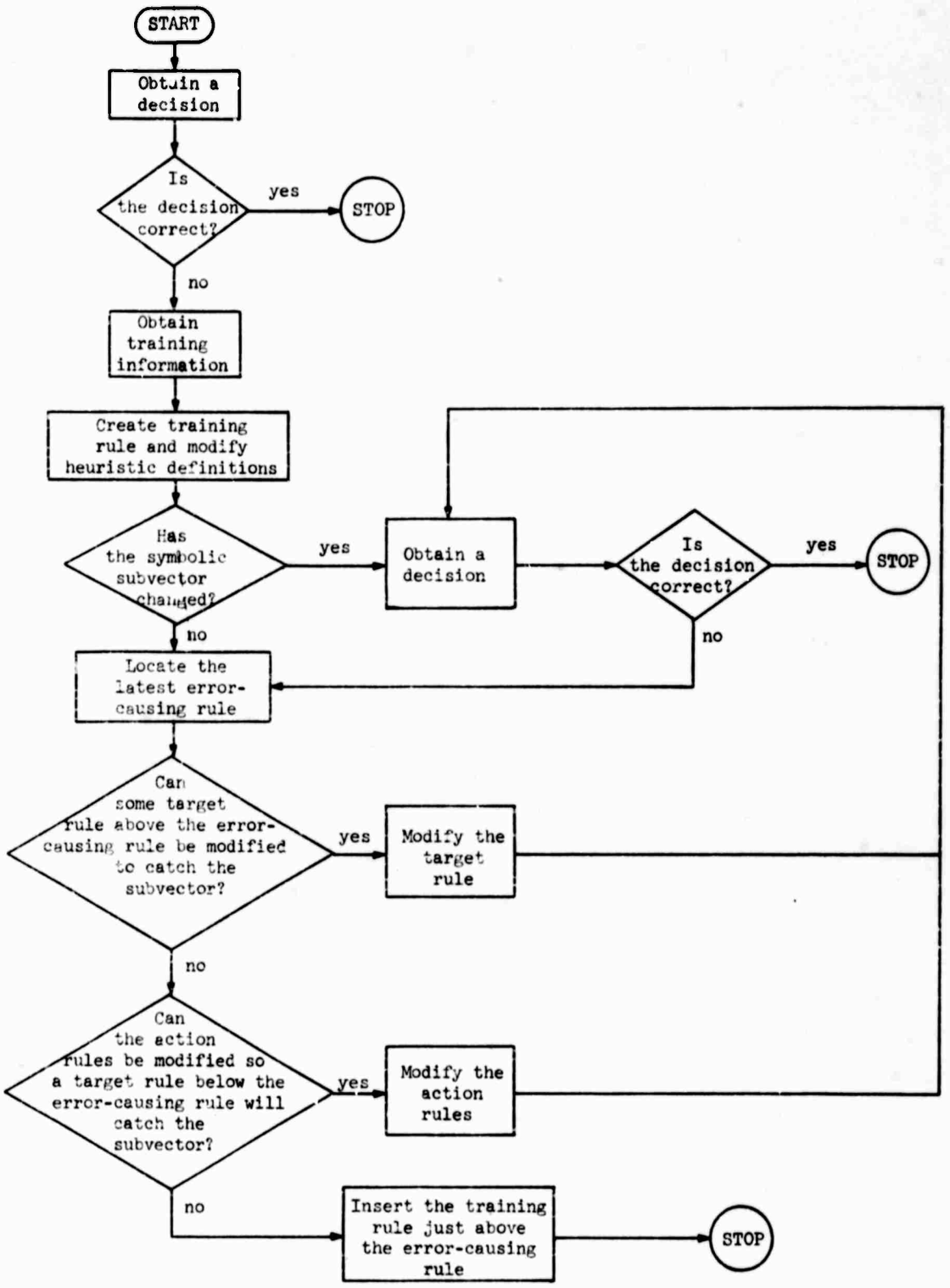


Figure 3-1. A block diagram of the training procedure.

(1) a good decision is "add 3 to the value of C ".

(2) the relevant variables are A and B .

(3) the decision is being made because " A is an A2 " and " B is small".

The training rule (constructed from the training information) is

$$(A2, B2, *) \rightarrow (*, *, c+3)$$

and the bf rules changed or created (on the basis of item (3) above) are

$$A1 \rightarrow A,\ A < 18$$

$$A2 \rightarrow A,\ A \geq 18$$

$$B2 \rightarrow B,\ B < 3 .$$

These bf rules change the symbolic subvector to (A2, B2, C1) which catches on rule 3. Thus the error-causing rule is rule 3. No action rules above or below the error-causing rule have the same form as the training rule, so the training rule is inserted into the list of action rules immediately above error-causing rule 3. The new set of rules is shown below. Here, when the program subvector is (18, 2, 11) the desired decision, "add 3 to the value of C ", is made.

1. $(A2, B1, *) \rightarrow (a+1, *, *)$
2. $(A1, *, C1) \rightarrow (*, b+2, *)$
3. $(A2, B2, *) \rightarrow (*, *, c+3)$
4. $(*, *, *) \rightarrow$ (random)
5. $A1 \rightarrow A\ \ A < 18$
6. $A2 \rightarrow A,\ A \geq 18$
7. $B1 \rightarrow B,\ B > 3$

8. $B2 \rightarrow B,\ B < 3$

9. $C1 \rightarrow C,\ C > 9$

MODIFYING A RULE ABOVE THE ERROR-CAUSING RULE. Let the program subvector at the beginning of training trial 2 be (12, 1, 7) . This parses to the symbolic subvector (A1, B2, C) which catches on rule 4 and leads to a random decision. Assume that this decision is unacceptable and that the training information is:

(1) a good decision is to "add 2 to the value of B ".

(2) the relevant variables are A and C .

(3) the decision is being made because " A is an A1 " and " C is large".

The training rule (constructed from the training information) is

$$(A1, *, C2) \rightarrow (*, b+2, *)$$

and the bf rule created (on the basis of item (3) above) is

$$C2 \rightarrow C,\ C > 6 .$$

This bf rule changes the symbolic subvector to (A1, B2, C2) which still catches on rule 4. Thus the error-causing rule is rule 4. Rule 2, above the error-causing rule, has the same form as the training rule and is suitable for modification to catch the symbolic subvector if $K \geq 3$. Let $K = 3$, then rule 2 is modified by replacing C1 with C2 . The new set of rules is shown below. Here, when the program subvector is (12, 1, 7) the desired decision, "add 2 to the value of B ", is made.

1. $(A2, B1, *) \rightarrow (a+1, *, *)$

2. $(A1, *, C2) \rightarrow (*, b+2, *)$

3. (A2, B2, *) → (*, *, c+3)

4. (*, *, *) → (random)

5. A1 → A, A < 18

6. A2 → A ≥ 18

7. B1 → B, B > 3

8. B2 → B, B < 3

9. C1 → C, C > 9

10. C2 → C, C > 6

MODIFYING A RULE BELOW THE ERROR-CAUSING RULE. Let the program subvector at the beginning of training trial 3 be (21, 4, 15) . This parses to the symbolic subvector ((A2), (B1), (C1,C2)) which catches on rule 1 and leads to the decision of incrementing A by 1 . Assume that this decision is unacceptable and that the training information is:

(1) a good decision is to "add 3 to the value of C ".

(2) the relevant variables are A and B .

(3) the decision is being made because " A is an A2 " and " B is small".

The training rule (constructed from the training information) is

(A2, B3, *) → (*, *, c+3)

and the bf rule created (on the basis of item (3) above) is

B3 → B, B < 5 .

This bf rule changes the symbolic subvector to ((A2), (B1,B3), (C1,C2)) which still catches on rule 1, making it the error-causing rule. Rule 3 below the error-causing rule has the same form as the training rule and

is suitable for modification to catch the symbolic subvector. Furthermore, the error-causing rule is suitable for modification to pass the subvector. Thus rule 3 is modified by replacing B2 with B3 , and rule 1 is modified by changing the definition of B1 to

$$B1 \rightarrow B,\ B > 4\ .$$

The new set of rules is shown below. Here, when the program subvector is (21, 4, 15) the desired decision, "add 3 to the value of C ", is made.

1. (A2, B1, *) → (a+1, *, *)
2. (A1, *, C2) → (*, b+2, *)
3. (A2, B3, *) → (*, *, c+3)
4. (*, *, *) → (random)
5. A1 → A, A < 18
6. A2 → A, A ≥ 18
7. B1 → B, B > 4
8. B2 → B, B < 3
9. B3 → B, B < 5
10. C1 → C, C > 9
11. C2 → C, C > 6

CONVERGENCE. The effectiveness of these modification techniques can be tested by using a program, rather than a human, as a trainer. The training program must contain a complete set of game heuristics in production rule form and must monitor the learning program, which initially contains no heuristics. Whenever the learning program makes a decision which conflicts with the one made by the training program, it will be

told by the training program the correct decision, the relevant variables, and why the decision was made. The training program's decisions are considered to be the correct decisions. If the modification techniques used were perfect for use in the task environment under consideration, the learning program would eventually grow a set of production rules leading to exactly the same decisions as the training program rules. Poor modification techniques would create a learning program which rarely made the same decision as the training program. Thus the speed and degree of convergence obtainable between the decisions generated by the learning program and those generated by the trainer can be used as a measure of the effectiveness of the modification and generalization procedures.

Applicability of Training Process

A pertinent question at this point is the following. Using the modification and generalization techniques just described what features of the task environment affect the speed and the degree of convergence obtainable between the decisions generated by the learning program and those generated by the training program? For the learning procedures even to be applicable each subvector variable must be considered to have a range consisting of a set of integer values. When this condition is satisfied convergence can be obtained, however the speed and degree of convergence depend upon the properties of the "decision space" utilized by the trainer.

DECISION SPACE. The decision space of the trainer is considered to be an n-dimensional space which has a dimension corresponding to each of

the n variables in the subvector. Thus each point in this space represents a game situation, and the entire space represents the set of all possible game situations.

The trainer is assumed to know the correct decision to make in every game situation, i.e., it has a decision associated with each point in its decision space. For example, let $\beta = (P, B)$ where P and B each have a range from 1 to 5 and where decisions d_1, d_2, d_3, and d_4 may be made. Then the decision space for the trainer could have the form shown below.

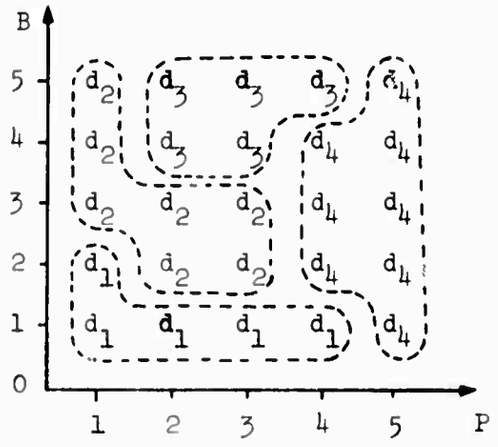


Figure 3-2.

The degree to which identical decisions tend to form groups will be called the clustering effect, indicated by the dotted lines in the above figure. In this example there is a high degree of clustering. An example of minimal clustering is shown below.

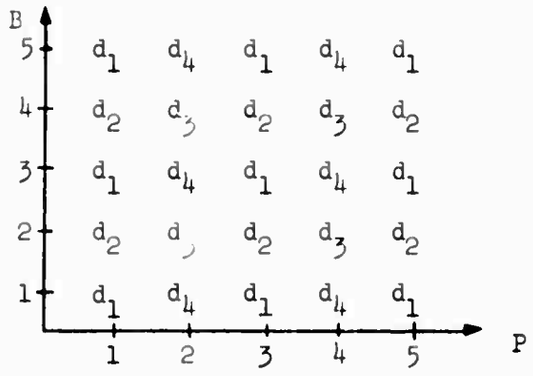


Figure 3-3.

SPEED OF CONVERGENCE. It can now be seen that the speed of convergence depends on the degree of clustering inherent in the decision space of the trainer. If there is a high degree of clustering then convergence will be rapid, that is, the learning system will be able to accurately imitate the training program after learning only a small number of action rules. If, however, there is a low degree of clustering, convergence will be slow. For example, with minimal clustering the system will not converge until it has acquired one action rule for each game situation in the entire decision space.

DEGREE OF CONVERGENCE. The degree of convergence obtainable from the learning system, on the other hand, depends on the degree of consistency exhibited by the trainer during the training process. If the trainer is very consistent in its task of supplying decisions when presented with game situations (i.e., the arrangement of decisions in its decision space is very stable) a high degree of convergence is possible.

3.3 LEARNING WITHOUT EXPLICIT TRAINING

In section 3.2 it was shown how heuristics in the form of production rules can be learned when the following information is available for each move or game decision made by the program:

(1) a good decision for the situation,

(2) the relevant situation elements, and

(3) the reason why the decision is being made.

Training is one way to provide the program with this information, but this technique requires the presence and participation of a trainer. Since humans can learn to play games without explicit training, developing programs which also can learn without explicit training seems a reasonable goal. This can be attained if the program itself can be made to generate the training information, either through logical deduction or hypothesis formation. Once the training information is generated the program can proceed as outlined in the previous section and in a sense train itself. One difficulty is that some mechanism must be included for testing the hypotheses formed and for eliminating useless ones. Further, this mechanism must be compatible with the generalization techniques used in the training process. A procedure will now be described which enables the program to generate the training information during the normal course of play and thus learn heuristics without explicit training.

AXIOMATIZATION. The fundamental problem at this point is: how can the program hypothesize reasonable heuristic rules without explicit training? The chance of finding a reasonable or useful heuristic by creating heuristic rules at random seems rather remote. A novel way to attack the problem is to formalize or axiomatize (McCarthy, 1959) the following for the

game under consideration:

(1) the rules of the game,

(2) statements (or "axioms") about the game,

(3) general statements about techniques used in game playing.

The result is a set of logical statements or premises, from which new statements can be deduced using rules of deductive inference. These new statements can then be used as the basis for creating new heuristic rules.

This technique of logical deduction can be used by the program to obtain item (1) of the training information, that is, a good decision for the given game situation. This process entails (a) making a decision in a situation S, (b) noting the effect on S of the subsequent decision by the opponent, and (c) using the information about S and the change in S together with the set of logical statements to deduce what the original decision should have been. It was noted in section 3.1 that the longer the sequence of decisions, the easier it is to evaluate the sequence as being good or bad. This technique of using logical deduction permits the evaluation of a decision sequence of the worst type, a sequence of length one. An example of this technique applied to a particular game, as well as a complete set of logical statements for the game, is presented in chapter 5.

DECISION MATRIX. Item (3) of the training information can be obtained from a decision matrix which is game dependent and is given to the program before learning starts. Each row of the matrix stands for a game decision or class of decisions and each column for a subvector variable. Each entry E_{ij} in the matrix indicates why the variable j is relevant, if when the decision i is made it is in fact relevant. For example,

if the program can determine that decision i is good and variable j is relevant, and entry E_{ij} is the term "large" then it knows that decision i was made because variable j is large. An underlying assumption here is that when a variable is relevant for a particular decision or class of decisions it is always relevant for the same reason. The types of reasons under consideration are simply (a) the category the current value of the variable belongs to (for exclusive variables), and (b) the magnitude indication associated with the current value of the variable (for overlapping variables).

A linear polynomial used to determine a move decision is somewhat analogous to a decision matrix with just one row but with one column for each parameter of the polynomial. The entries in the matrix would all be the term "large", since whenever a decision is picked it is always because the relevant parameters are large and thus increase the value of the polynomial. Another heuristic program which is supplied with information in matrix form is GPS (Newell, Shaw, and Simon, 1959). This program relies on a connection table to provide information about the operators relevant to reducing certain differences.

HYPOTHESIS FORMATION. Item (2) of the training information can be obtained through the generation and testing of hypotheses concerning the relevancy of subvector variables. Again the problem of generating useful or reasonable hypotheses arises. This problem can be solved for the special case of relevancy hypotheses in the following manner. Let the initial hypotheses in every case be that all subvector variables are relevant; this means that the left parts of the training rules constructed from the 3 items of training information will initially contain no * 's. Testing

will consist of noting whether or not a particular training rule (placed in the set of action rules by step 7 of the training procedure) catches the symbolic subvector when the action advocated by the rule is determined to be the correct decision. If the rule does not catch the subvector, the hypothesis for that rule concerning the relevancy of the variables is changed by making some of the variables in the left part of the rule irrelevant. This makes the rule more general since it then applies to a greater variety of situations.

This technique can be easily incorporated into the training procedure as follows. If it is desired to modify an hypothesized action rule to catch the subvector and the rule cannot be suitably modified by replacing symbolic values then the following action is taken. The left part of the rule is modified by making a minimum number of variables irrelevant while still increasing the generality enough so the rule can catch the symbolic subvector. Of course some limit must be imposed on the degree of generality which may be obtained, otherwise the hypothesized action rules would eventually contain all * 's in their left parts. Let N stand for the minimum allowable number of variables which must remain relevant in the left part of an action rule. Then, when an hypothesized action rule has only N symbolic values which are not * 's in its left part it cannot be modified by reducing the number of its relevant variables. The value of N depends on the number of subvector variables used and the particular game under consideration.

Revised Training Procedure

The technique just described can be merged with the training procedure outline in section 3.2 by making a few minor changes. This revised training procedure outline is shown below.

1. a. Parse the program subvector to obtain the symbolic subvector.

   b. Drop the symbolic subvector through the action rules to obtain a decision.

   c. If the trainer indicates that the decision was acceptable then stop, otherwise go to step 2.

2. a. Obtain the training information from the trainer.

   b. Construct an action rule (to be called the training rule) from this information.

   c. Use item (3) of the training information to change or create bf rules which represent heuristic definintions. If this changes the symbolic subvector then go to step 3, otherwise go to step 4.

3. a. Drop the new symbolic subvector through the action rules to obtain a decision.

   b. If the decision is the one advocated by item (1) of the training information then stop, otherwise go to step 4.

4. a. Locate the action rule responsible for the unacceptable decision made in step 3 (or in step 1 if step 3 was skipped). This action rule will be called the error-causing rule.

5. a. Search the action rules above the error-causing rule for a non-hypothesized rule which has the same form as the training rule and is suitable for modification to catch the symbolic subvector. This rule will be called the target rule.

b. If such a rule is found use the training generalization techniques to modify it to catch the symbolic subvector and go to step 3, otherwise search the action rules above the error-causing rule for an hypothesized action rule leading to the decision advocated by the training information. If such a rule is found, modify it to catch the subvector by making a minimum number of variables irrelevant if this can be done and still leave N variables relevant and go to step 3; if no action rules suitable for this type of modification can be found above the error-causing rule then go to step 6.

6. a. Search the action rules below the error-causing rule for a non-hypothesized rule which has the same form as the training rule and is suitable for modification to catch the symbolic subvector. This rule will be called the target rule.

b. If (1) such a rule if found, (2) the error-causing rule is suitable for modification to pass the symbolic subvector, and (3) the rules between the error-causing rule and the target rule either pass the symbolic subvector or are suitable for modification to pass it then use the training generalization techniques to modify the target rule to catch the subvector, the error-causing rule to pass the subvector and go to step 3, otherwise go to step 7.

7. a. Place the training rule immediately above the error-causing rule in the list of action rules and stop.

An example of the operation of the revised training procedure will now be given for a state vector composed of overlapping variables A, B, and C . It will be assumed that K = 3 , N = 1 , and the decision matrix is:

	d_1	d_2	d_3
A	large	large	small
B	small	large	small
C	small	small	large

Figure 3-4.

where d_1 stands for "add 1 to the value of A ", d_2 stands for "add 2 to the value of B " and d_3 stands for "add 3 to the value of C ". The initial set of production rules for this example is shown below.

1. (A1, *, C1) → (*, *, c+3)
2. (*, *, *) → (random)
3. A1 → A, A > 10
4. C1 → C, C < 15

INSERTING AN HYPOTHESIZED ACTION RULE. Let the program subvector be (15, 12, 2) . This parses to (A1, B, C1) which catches on rule 1 and leads to the decision of incrementing C by 3 . The opponent now makes a decision and the program uses the information about the resulting game situation to logically deduce what its own decision should have been. Assume that the program deduces that a good decision would have been "add 2 to the value of B ". The training rule is then

$$(A2, B1, C2) \rightarrow (*, b+2, *)$$

and the bf rules changed or created are

$$A2 \rightarrow A,\ A > 14$$

$$B1 \rightarrow B,\ B > 11$$

$$C2 \rightarrow C,\ C < 3$$

Since no rules in the set of action rules lead to the correct decision the training rule is inserted above the error-causing rule (rule 1) as specified in step 7 of the revised training procedure outline. In this case the training rule is an hypothesized rule and is marked in some way so the program can distinguish it from action rules which were not hypothesized. The new set of rules is shown below. Here, when the program subvector is (15, 12, 2) the desired decision, "add 2 to the value of B " is made.

1. $(A2, B1, C2) \rightarrow (*, b+2, *)$ hypothesized
2. $(A1, *, C1) \rightarrow (*, *, c+3)$
3. $(*, *, *) \rightarrow$ (random)
4. $A1 \rightarrow A,\ A > 10$
5. $A2 \rightarrow A,\ A > 14$
6. $B1 \rightarrow B,\ B > 11$
7. $C1 \rightarrow C,\ C < 15$
8. $C2 \rightarrow C,\ C < 3$

MODIFYING AN EXISTING HYPOTHESIZED RULE. Let the program subvector at the time of the program's next move decision be (18, 13, 14) . This parses to ((A1, A2), (B1), (C1)) which catches on rule 1 and

leads to the decision of incrementing C by 3 . The opponent now make a decision, and the program logically deduces what its own decision should have been. Assume that the program deduces that a good decision would have been "add 2 to the value of B ". The training rule is then

$$(A2, B1, C1) \rightarrow (*, b+2, *)$$

and no bf rules are changed or created. Rule 1 which leads to the correct decision and is above the error-causing rule cannot be modified to catch the subvector by replacing symbolic values since K is too small. However, this rule is an hypothesized one and can therefore be modified by making variables irrelavant. In this case only the variable C must be considered irrelevant, so rule 1 becomes

$$(A2, B1, *) \rightarrow (*, b+2, *) .$$

The new set of rules is shown below.

1. $(A2, B1, *) \rightarrow (*, b+2, *)$ hypothesized
2. $(A1, *, C1) \rightarrow (*, *, c+3)$
3. $(*, *, *) \rightarrow$ (random)
4. $A1 \rightarrow A, A > 10$
5. $A2 \rightarrow A, A > 14$
6. $B1 \rightarrow B, B > 11$
7. $C1 \rightarrow C, C < 15$

Here when the program subvector is (18, 13, 14) the desired decision, "add 2 to the value of B " is made.

COMBINING TRAINING AND HYPOTHESIS FORMATION. The system just described

can learn heuristics in a variety of ways. It can learn through

(1) training alone: here the action rules are non-hypothesized, since they are all based on information obtained from a trainer,

(2) hypothesis formation alone: here the action rules are all hypothesized, or

(3) training and hypothesis formation combined: here the action rules are a mixture of hypothesized and non-hypothesized rules.

In any case the program starts with no heuristic definitions and just one heuristic rule, (*, *, *) → (random) , which tells it to initially make decisions at random. Training and hypothesis formation may be combined by first giving the program a number of explicit training trials and then letting it learn through hypothesis formation during actual game play. In this situation the hypothesized action rules must be distinguished from the non-hypothesized ones since the two types of rules require different generalization techniques. However, when an hypothesized rule is generalized to the extent of having only N variables remaining in its left part it can be given the status of a non-hypothesized rule.

Creation of Redundant Action Rules

The use of hypothesized action rules increases the possibility of accidentally creating redundant action rules. These are rules which can be removed from the list of action rules without in any way affecting the decisions made by the system.

TYPES OF REDUNDANCIES. Two types of redundancies will be considered:

(a) subordinate redundancy, where a rule in the ordered list causes a rule below it to be redundant, and (b) superordinate redundancy, where a rule in the ordered list causes a rule above it to be redundant. To illustrate, let rule i be above rule j in the list of action rules. Then rule i makes rule j a subordinate redundant rule if i keeps j from ever catching a symbolic subvector, by itself catching all generated subvectors that could otherwise be caught by j. This situation occurs when each symbolic value in the left part of rule i defines a set which includes the set defined by the corresponding symbolic value of rule j.

Conversely, rule i is a superordinate redundant rule if every symbolic subvector caught by i would be caught by another rule below i leading to the same decision as i if rule i were removed. This situation occurs when each symbolic value in the left part of a lower rule j defines a set which includes the set defined by the corresponding symbolic value of rule i, and rule i, rule j, and all rules between i and j lead to the same decision.

EXAMPLE. As an example, consider the set of production rules shown below, where the state vector contains overlapping variables A, B, and C, and 3 different decisions are denoted by d_1, d_2, and d_3.

1. (A1, B1, *) → d_1
2. (A2, B2, C1) → d_2
3. (*, B2, C2) → d_3
4. (*, B1, *) → d_3
5. A1 → A, A > 5
6. A2 → A, A > 10

7. $B1 \rightarrow B,\ B < 9$

8. $B2 \rightarrow B,\ B < 4$

9. $C1 \rightarrow C,\ C > 15$

10. $C2 \rightarrow C,\ C < 7$

Here rule 1 makes rule 2 a subordinate redundant rule, and rule 4 makes rule 3 a superordinate redundant rule. As a consequence, the set of production rules shown below, with action rules 2 and 3 removed, is exactly equivalent to the original set.

1. $(A1, B1, *) \rightarrow d_1$

2. $(*, B1, *) \rightarrow d_3$

3. $A1 \rightarrow A,\ A > 5$

4. $B1 \rightarrow B,\ B < 9$

Note that the removal of action rules 2 and 3 made bf rules 6, 8, 9, and 10 superfluous and thus led to their removal also.

REDUNDANCY CHECKS. In a learning system of the type proposed in this section redundancy checks should be made periodically to keep the action rule list from becoming too long. However, the danger in removing redundancies before learning is completed is that rules may be removed which later would have been generalized upon and made non-redundant. Premature removal of this type will tend to slow down the learning process. Thus both the length of the action rule list and the speed of convergence of the learning system must be considered when determining how often redundancy checks should be made.

CHAPTER 4

IMPLICATIONS FOR S-R THEORIES OF LEARNING

4.1. INTRODUCTION

In psychology, learning theories fall into two major categories, stimulus-response (S-R) theories and cognitive theories (Hilgard and Bower, 1966). The stimulus response theories view learning as the acquisition of stimulus-response chains or "habits". Organisms are assumed to merely learn responses, and to resort to trial and error when confronted with a novel problem for which no response has been learned. Cognitive theories on the other hand, view learning as the acquisition of memories or expectations in the form of cognitive structures. Organisms are assumed to learn facts, and to employ "insight" based on the understanding of the essential relationships involved when confronted with a novel problem.

In both categories, model building has proved to be a useful technique for describing data and predicting experimental results. Mathematical models of learning (Bush and Mosteller, 1955; Estes, 1959) have been constructed which are simple, concise descriptions of quantitative data, many capable of yielding quite accurate numerical predictions. As Bower (1966) points out, most of the theoretical work in mathematical learning theory has been in the area of "stimulus-response associationism", although cognitive theories can be and often are expressed in mathematical form.

More recently, information-processing models of human behavior and intelligence have emerged (Feigenbaum, 1959; Feldman, 1959; Newell

and Simon, 1961; Hunt, 1962; Simon and Kotovsky, 1963; Reitman, 1965). This type of model, in the form of a computer program, can be regarded as a theory of the psychological processes underlying the behavior being simulated. The information-processing model is a precise, unambiguous statement of the theory and is well suited for generating explicit predictions.

Up to now S-R theories have been used to explain many types of simple learning, but not processes as complex as strategy or heuristic learning. The information-processing system described in Chapter 2 and 3 suggests a number of approaches to the problem of constructing S-R theories or models of human strategy learning in game-playing or problem-solving environments. Some of the possible approaches to this problem will now be examined and evaluated.

4.2. AN S-R INTERPRETATION OF PRODUCTION RULES

A production rule defining the change to make in the state vector $\mathcal{E}$ of a program has the form:

$$(A1, B1, C1) \rightarrow (f_1(\mathcal{E}), f_2(\mathcal{E}), f_3(\mathcal{E})) ,$$

where A1, B1, and C1 are symbolic representations of the current values of the subvector, and $f_1(\mathcal{E})$, $f_2(\mathcal{E})$ and $f_3(\mathcal{E})$ are functions or arithmetic expressions defining the new values for the subvector. It will be recalled that the subvector is the set of program variables which may influence or be affected by the decisions of the program. Another way to interpret the subvector is to consider it a description of a particular game situation, where each element of the subvector is a value of a pertinent attribute of the situation. The production rule shown above can thus be thought of as a situation-action pair

$$S \rightarrow A$$

which effectively means "in situation S take action A". Under this interpretation, strategy learning simply consists of the acquisition of S-A pairs.

S-R Models of Strategy Learning

Models of human strategy learning in a game-playing environment will now be proposed. These models learn by being presented with a series of game situations, the corresponding actions to take in each situation, and the reason why each action is taken. A situation description consists of a list of all pertinent aspects of the situation, each aspect being called a situation (or stimulus) element.

CONSTRAINTS. All the models under consideration are based on certain constraints about how strategy learning can actually take place. The constraints thus postulated are the following:

1. Association: the stimulus elements of a situation become associated with or connected to the correct action to take in that situation.
2. One-trial learning: the stimulus elements are connected completely to an action after one training trial.
3. Dependent elements: a situation description is a pattern of dependent stimulus elements, i.e., the pattern, rather than the individual elements, becomes connected to the action.
4. Interference: the only way that forgetting can occur is through interference, that is, by replacing the action part, A , of an S-A connection with a new action A' .
5. Consistent training: the situation-action pairs presented to the model will not contain conflicting information, such as the same situation paired with two or more different actions. The effect of this constraint is that interference (and hence forgetting) will not occur.

Association, one-trial learning, and interference are postulated because they provide the models with a basic structure that is relatively simple. Dependent elements must be postulated, since in a game-playing situation the stimulus elements are quite highly interdependent. Consistent training is postulated so that complications due to forgetting may be neglected.

ACTUAL ELEMENTS. In a game-playing situation the pattern of stimulus elements that describes the situation at a particular time is composed of the values of the pertinent attributes of the situation. It is assumed that these values can be represented as integers. For example, consider a game with attributes H , P , and B , each having values from 1 to 10 . Then a typical situation description (pattern of stimulus elements) might be 2,9,5 meaning that this situation is defined by H having a value of 2 , P a value of 9 , and B a value of 5 . An asterisk as an attribute value indicates that the attribute may take on any value. Hence 6,*,4 represents a class of situations where H has the value 6 , P any value from 1 to 10 , and B the value 4 . These integer stimulus elements are called "actual" elements.

ABSTRACT ELEMENTS. Another type of element to be considered is the symbolic stimulus element, such as h1 , p1 , or b1 , where each symbol represents any element from a particular subset of integers. Thus h1,p1,b1 is a description of a class of situations. These symbolic stimulus elements are called "abstract" elements and are defined by partitioning the ranges of the attributes either into mutually exclusive and exhaustive subsets or into overlapping subsets. An example of the former type of partitioning for H is "h1: $H \leq 6$ and h2: $H > 6$". An example of the latter type is "h1: $H < 7$ and h2: $H > 3$".

STORAGE. If a pattern of stimulus elements S is presented to a model and the model fails to predict the correct action A , the model

is told the correct action, and the S-A connection is stored in a list. The storage process may consist of simply placing the new connection at the end of the previously learned connection list. If exclusive abstract elements are used, storage may consist of also growing a decision tree from the previously learned S-A connections. Furthermore, when overlapping abstract elements are present, storage may consist of the following steps.

(1) The definitions of the abstract elements are changed such that the new S-A connection is effectively placed in the previously learned connection list.

(2) If step (1) is not possible, the new S-A connection is added to the previously learned list by placing it immediately above the connection which led to the last error.

RETRIEVAL. When a model is given a situation description S , it must predict what action to take. It is assumed that this prediction is based in some way on the result of a retrieval process. The most elementary process consists of matching S against every situation description stored and if a perfect match is found retrieving the associated action. If no match is found an action is picked at random for output.

A more complicated process consists of comparing S to every situation description stored and choosing as the prediction the action associated with the description that is closest to S . Here closeness is defined as the distance between two descriptions, where a description, for n attributes, is thought of as a point in n-dimensional space.

A third possible process consists of filtering S down a decision tree or discrimination net grown from previously learned S-A connections. The action associated with the terminal node finally reached by S is then used as the prediction.

DEGREES OF FREEDOM. The preceding remarks concerning methods of representation, storage, and retrieval for the models will now be summarized. The models are permitted the following degrees of freedom:

1. Situation Representation
   a. Actual Elements (example: 9,4,7)
   b. Abstract Elements (example: h1,p2,b3)
      (1) Mutually exclusive definitions (example: h1: $H < 5$, h2: $H \geq 5$)
      (2) Overlapping definitions (example: h1: $H > 7$, h2: $H < 15$)
2. Storage Mechanism (storage of an S-A connection)
   a. Simple Placement: the connection is added to the end of the connection list already learned.
   b. Induction: a decision tree is grown based on the current list of learned S-A connections.
   c. Complex Placement: definitions of abstract elements are changed, if possible, to effectively place the connection in the learned list. Otherwise the connection is added just above the connection that led to the last error.
3. Retrieval Mechanism (retrieval of an A when given an S)
   a. Simple Search: the S is compared to all descriptions in the learned connection list, and if an exact match is

found the corresponding A is retrieved, otherwise an A is picked at random.

b. Stimulus Generalization: the S is compared to all descriptions in the learned connection list, and for the best match (defined by closeness in n-dimensional space) the corresponding A is used.

c. Tree-sorting: the S is sorted down a decision tree to a terminal node, and the A at that node is used.

FEASIBLE MODELS. Allowing the preceding degrees of freedom should permit the construction of 3 x 3 x 3 or 27 different models. Actually only 10 of these models are feasible due to certain incompatibilities which exist between the proposed methods of representation, storage and retrieval. In the diagram shown below each square represents one of the 27 hypothetical models. The X's indicate which of these are the 10 feasible models.

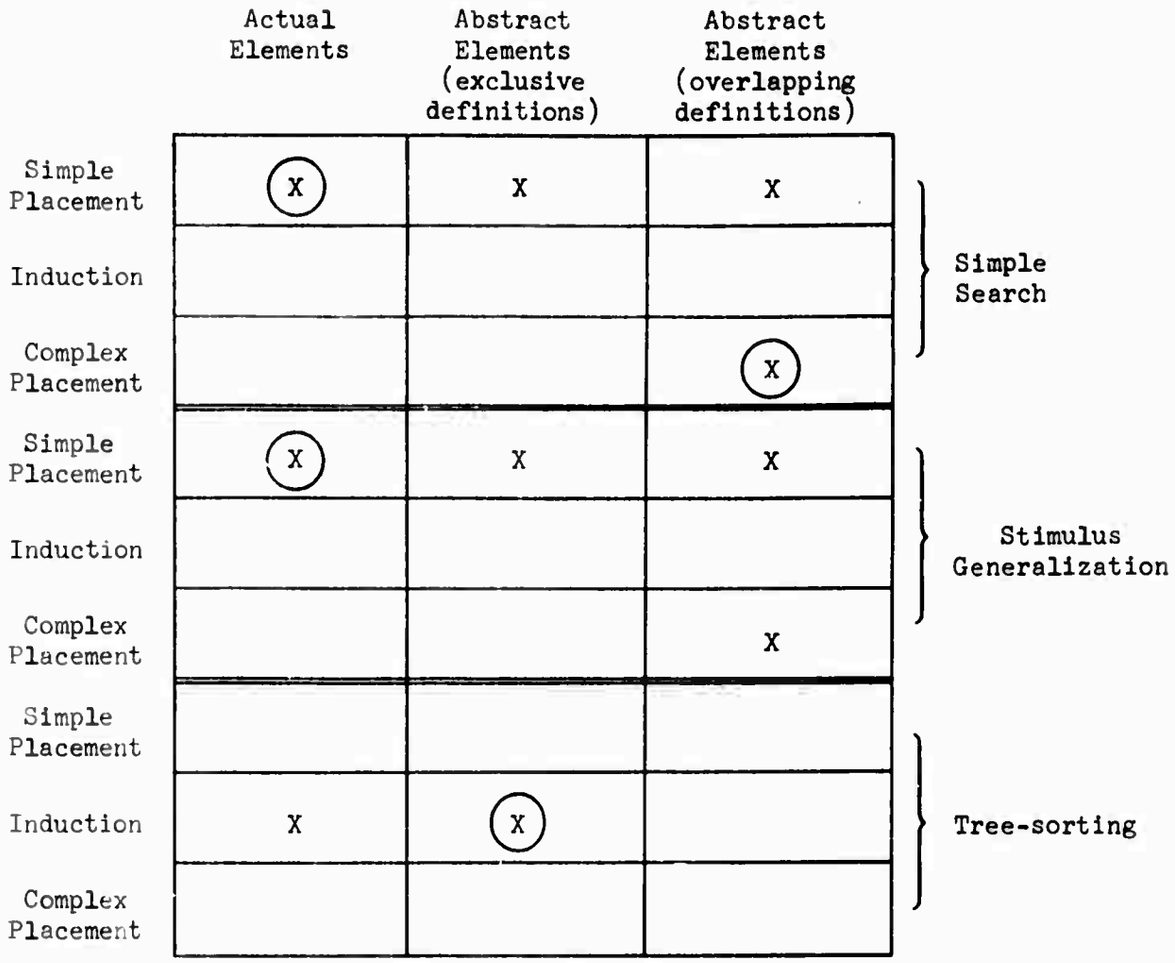


Figure 4-1.

Four of these models, indicated by the circles in Figure 4-1, will be described in this chapter and their operation illustrated by the training sequence given in Figure 4-2.

TRAINING. Training consists of supplying the models with training information after each error. This training information consists of (1) the correct decision, (2) the elements relevant to making the correct decision, and (3) the reason why the decision is being advocated, expressed in terms of an evaluation of each relevant element. If a model uses

actual elements, item (3) is not required since there are no definitions to learn. If a model uses abstract elements, item (3) is necessary, and the model is assumed to learn the definitions of these elements using the procedure outlined in section 3.2. Figure 4-2 gives the definitions the models would learn if this procedure were applied to the training sequence shown. Model operation will be illustrated as though the models are given these definitions, in order to simplify the examples presented. However, in an actual experimental design the models would be required to learn the definitions.

Range of Actual Values:	H(1-50)	P(1-60)	B(1-10)
Mutually Exclusive Definitions:	h1(H>25)	p1(P>9)	b1(B>7)
	h2(10<H≤25)	p2(P≤9)	b2(B≤7)
	h3(H≤10)		
Overlapping Definitions:	h1(H<16)	p1(P>20)	b1(B<7)
	h2(H<5)	p2(P<9)	b2(B>9)
	h3(H>36)		

Training Sequence:

	situation description	correct decision	relevant elements	reason
1.	15,21,6	A3	H,P,B	H is "h2" or "small", P is "p1" or "large", B is "b2" or "small"
2.	4,28,3	A4	H	H is "h3" or "small"
3.	13,8,4	A2	H,P,B	H is "h2" or "small", P is "p2" or "small", B is "b2" or "small"
4.	37,4,9	A1	H,P	H is "h1" or "large", P is "p2" or "small"
5.	12,9,10	A4	H,B	H is "h3" or "small", B is "b1" or "large"
6.	1,42,17	A4	H	H is "h3" or "small"
7.	12,5,5	A2	H,P,B	H is "h2" or "small", P is "p2" or "small", B is "b2" or "small"

Figure 4-2. Training sequence and definitions to illustrate model operation.

A Simple Model

The first model to be described is defined as having the following characteristics:

(1) actual elements,

(2) simple placement,

(3) simple search.

This is called a Simple Model and is the most elementary one which can be constructed within the framework just proposed. Its operation will be illustrated for the first five trials of the training sequence shown in Figure 4-2.

PREDICTION. When the model is given a situation description S and is asked to predict A it matches S against all left sides of the connections in the learned list going from top to bottom until an exact match is found. The right side of the connection whose left side exactly matches S is then used as the prediction. If the prediction is wrong, a new connection, formed from S and the correct action, is added to the bottom of the list of learned connections.

The model is assumed to initially consist of a single S-A connection of the form

$$*,*,* \rightarrow \{\text{action picked at random}\}$$

which catches all situation descriptions and leads to an action being picked at random from the set of possible actions. Since the model learns through training what actions are possible, on the first trial the known set of possible actions is empty and no prediction is made.

OPERATION. The operation of the Simple model for the first five training trials is depicted below.

	S	Learned S-A Connections	Predicted A	Correct A
1.	15,21,6	*,*,* → { }	none	A3
2.	4,28,3	15,21,6 → A3 *,*,* → {A3}	A3 (from last connection)	A4
3.	13,8,4	15,21,6 → A3 4,*,* → A4 *,*,* → {A3,A4}	A4 (from last connection)	A2
4.	37,4,9	15,21,6 → A3 4,*,* → A4 13,8,4 → A2 *,*,* → {A2,A3,A4}	A3 (from last connection)	A1
5.	12,9,10	15,21,6 → A3 4,*,* → A4 13,8,4 → A2 37,4,* → A1 *,*,* → {A1,A2,A3,A4}	A2 (from last connection)	A4

EVALUATION OF THE MODEL. Because of the wide range of values of the three attributes, the probability of finding an exact match for S among the learned connections is quite small, especially if the situation descriptions are chosen at random. Hence the model does little more than make a random guess when presented with an A and asked for a prediction. This model is clearly too simple to serve as a useful theory of human strategy learning.

A Stimulus Generalization Model

The second model to be described is called the Stimulus Generalization model and is defined as having the following characteristics:

(1) actual elements,

(2) simple placement,

(3) stimulus generalization.

The operation of this model will be illustrated for the entire training sequence given in Figure 4-2.

PREDICTION. The model makes a prediction, when given a situation description S, be comparing S to every situation description stored in the learned connection list and choosing as the prediction the action associated with the description that comes closest to matching S. Closeness is defined as the distance between two descriptions when each description, for n attributes, is interpreted as a point in n-dimensional space. However, descriptions containing one or more *'s must be thought of as hyperplanes in the n-dimensional space. For example, if n=3 then 15,21,6 represents a point, 15,*,6 a line, and 15,*,* a plane in 3-dimensional space. If the prediction made by the model is wrong, a new connection composed of S and the correct action is added to the end of the learned connection list. No prediction is made on the first trial since at this point the connection list is empty.

OPERATION. The operation of the Stimulus Generalization model for the training sequence of Figure 4-2 is shown below.

	S	Learned S-A Connections	Distance Between S and Connection	Predicted A	Correct A
1.	15,21,6	none	none	none	A3
2.	4,28,3	15,21,6 → A3	13.4	A3	A4
3.	13,8,4	15,21,6 → A3 4, *,* → A4	13.3 9.0	A4	A2
4.	37,4,9	15,21,6 → A3 4, *,* → A4 13,8,4 → A2	28.0 33.0 24.8	A2	A1
5.	12,9,10	15,21,6 → A3 4, *,* → A4 13,8,4 → A2 37,4,* → A1	13.0 8.0 6.2 25.5	A2	A4
6.	1,42,17	15,21,6 → A3 4, *,* → A4 13,8,4 → A2 37,4,* → A1 12,*,10 → A4	27.6 3.0 38.4 52.3 13.1	A4	A4
7.	12,5,5	15,21,6 → A3 4, *,* → A4 13,8,4 → A2 37,4,* → A1 12,*,10 → A4	16.3 8.0 3.3 25.0 5.0	A2	A2

The model always chooses an A such that the distance between S and the left side of the connection containing A is minimized. In trial 5, for instance, action A2 is predicted by the model because the distance d between S (12,9,10) and the situation description of the third connection (13,8,4) is the smallest. This calculation is illustrated below.

$$d = \sqrt{(x_1-x_2)^2 + (y_1-y_2)^2 + (z_1-z_2)^2}$$

$$= \sqrt{(12-13)^2 + (9-8)^2 + (10-4)^2} \cong 6.2$$

A "*" is considered to be an exact match for any value when the above formula is used to calculate d .

EVALUATION OF THE MODEL. This model is clearly superior to the Simple model since the closest match to S is always found, and thus the model need not resort to random predictions. However, this model does have its weak points. First, the type of comparison procedure suggested for retrieval is quite involved, and it is difficult to imagine humans actually performing such mathematically-oriented calculations when placed in such a training situation. Second, in the early stages of training virtually every training trial adds a new S-A connection to the learned list. Since the input S must always be compared with every connection on this list, the time needed to retrieve a response (i.e., the latency) sharply increases as the number of reinforced trials increases.

An Induction Model

The third model to be described is the Induction model, which is defined as having the following characteristics:

(1) abstract elements with mutually exclusive definitions,

(2) induction,

(3) tree-sorting.

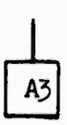
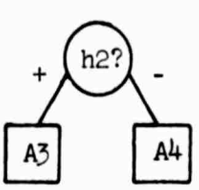
The training sequence and definitions in Figure 4-2 will be used to illustrate the operation of this model.

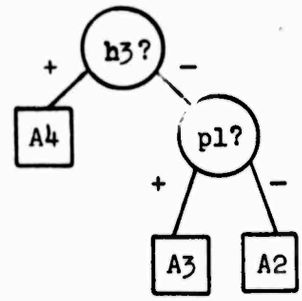
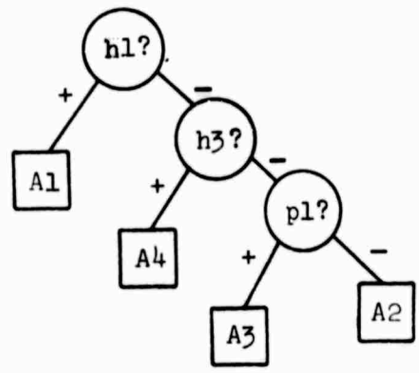
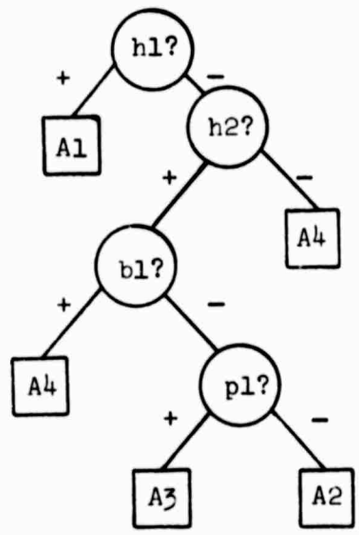
PREDICTION. The Induction model makes a prediction by sorting the given S to a terminal node in a decision tree previously grown using the current list of learned S-A connections. The action associated with that terminal node is used as the prediction. If the prediction is

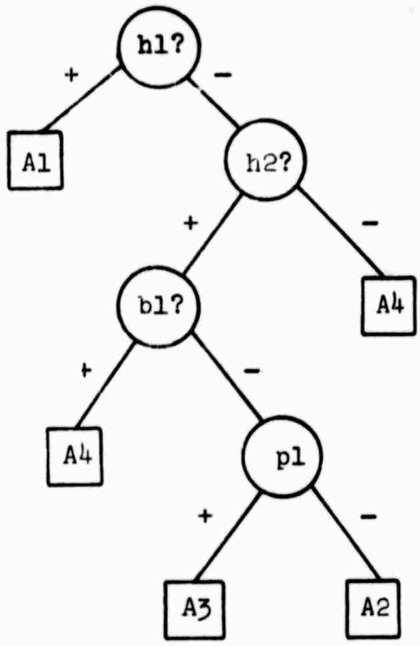
wrong, the connection formed by S and the correct action is added to the learned S-A connection list, and a new tree is grown.

The generalization technique used to grow the tree is an extension of the technique used by Hunt (1962,1966) for growing concept trees, that is, trees for distinguishing between positive and negative instances of a concept. The decision tree partitions the universe of situations into m sets, one for each possible action that may be taken. Each situation element is considered to be an attribute of the situation, and the tests made at the nodes of the decision tree are tests on the possible values of these attributes. The tree-growing technique is summarized in Appendix A, Part I.

OPERATION. The operation of the Induction model for the training sequence in Figure 4-2 will now be illustrated. No prediction is made on the first trial since at this point no decision tree exists.

	S	Learned S-A Connections	Tree used to produce a prediction	Predicted A	Correct A
1.	15,21,6 h2,p1,b2	none	none	none	A3
2.	4,28,3 h3,p1,b2	h2,p1,b2 → A3		A3	A4
3.	13,8,4 h2,p2,b2	h2,p1,b2 → A3 h3,*,* → A4		A3	A2

	S	Learned S-A Connections	Tree used to produce a prediction	Predicted A	Correct A
4.	37,4,9 h1,p2,b1	h2,p1,b2 → A3 h3,*,* → A4 h2,p2,b2 → A2		A2	A1
5.	12,9,10 h2,p2,b1	h2,p1,b2 → A3 h3,*,* → A4 h2,p2,b2 → A2 h1,p2,* → A1		A2	A4
6.	1,42,17 h3,p1,b1	h2,p1,b2 → A3 h3,*,* → A4 h2,p2,b2 → A2 h1,p2,* → A1 h2,*,b1 → A4		A4	A4

		Learned S-A Connections	Tree used to produce a prediction	Predicted A	Correct A
7.	l2,5,5	h2,p1,b2 → A3		A2	A2
	h2,p2,b2	h3,*,* → A4			
		h2,p2,b2 → A2			
		h1,p2,* → A1			
		h2,*,b1 → A4			

Note that a completely new tree must be grown each time another S-A connection is added to the learned list. Only in trial 7 above was a new tree unnecessary, since the correct prediction was made in trial 6 and consequently no S-A connection was added to the list.

EVALUATION OF THE MODEL. The Induction model is possibly superior to the models previously presented since it does not have to resort to random predictions and the retrieval mechanism is somewhat more satisfying as an explanation of human cognition. Also, this model does not lead to a sharp increase in response retrieval time when the number of reinforced training trials increases, as does the Stimulus Generalization model. This is true because (a) the response retrieval time depends entirely on the time needed to sort the S down the tree, and this

sorting time increases very slowly as the size of the tree increases; the retrieval time doesn't depend on the time needed to grow the tree since tree growing occurs at the end of a trial, as part of the storage process, and (b) fewer S-A connections are stored during training to a criterion of say x correct trials in a row, and fewer connections means faster retrieval.

Although this model is possibly superior to the others, it does have its deficiencies. First, the decision tree that is grown, and hence the action retrieved, is highly dependent on the algorithm used to determine which attribute value is to be chosen as a test at a node, and it is not clear what the best algorithm is. However, this dependency can be turned into a virtue if one can see how to modify the algorithm to improve the performance of the model. Second, the model must be presented with completely consistent training information in order to function properly. If during training it is given information implying that more than one action is possible in a certain situation, the tree-generating mechanism will generate some branches which never terminate. For example, if the model is told the S-A connections h1,p1,* → A1 , and h1,*,b2 → A2 are both valid it will grow a non-terminating branch. This feature is a deficiency because humans are able to learn strategies even when presented with inconsistent information.

A Complex-placement Model

The last model to be described is the Complex-placement model, which is defined as having the following characteristics:

(1) abstract elements with overlapping definitions,

(2) complex placement,

(3) simple retrieval.

The operation of this model will be illustrated for the training sequence and definitions given in Figure 4-2.

PREDICTION. The Complex-placement model makes a prediction by comparing the given S to all situation descriptions in the learned connection list, going from top to bottom, and if an exact match is found the corresponding A is retrieved. If a match is not found, an action is selected at random from the known set of possible actions. When an incorrect action is retrieved the abstract definitions are changed, if possible, to effectively place the connection formed by S and the correct A in the existing list. Otherwise this new connection is added to the existing ordered connection list immediately above the S-A connection that led to the previous error. Initially, the model consists of a single S-A connection which catches all S's and leads to an action being picked at random, as in the Simple model.

OPERATION. The operation of the Complex-placement model for the training sequence of Figure 4-2 is shown below.

	S	Learned S-A Connections	Predicted A	Correct A
1.	15,21,6 h1,p1,b1	*,*,* → { }	none	A3
2.	4,28,3 h1-h2,p1,b1	h1,p1,b1 → A3 *,*,* → {A3}	A3 (from the first connection)	A4

	S	Learned S-A Connections	Predicted A	Correct A
3.	13,8,4	h2,*,* → A4	A4	A2
	h1,p2,b1	h1,p1,b1 → A3	(from last connection)	
		,,* → {A3,A4}		
4.	37,4,9	h2,*,* → A4	A2	A1
	h3,p2,b	h1,p1,b1 → A3	(from the last connection)	
		h1,p2,b1 → A2		
		,,* → {A2,A3,A4}		
5.	12,9,10	h2,*,* → A4	A3	A4
	h1,p,b2	h1,p1,b1 → A3	(from last connection)	
		h1,p2,b1 → A2		
		h3,p2,* → A1		
		,,* → {A1,A2,A3,A4}		
6.	1,42,17	h2,*,* → A4	A4	A4
	h1-h2,p1,b2	h1,p1,b1 → A3	(from first connection)	
		h1,p2,b1 → A2		
		h3,p2,* → A1		
		h1,*,b2 → A4		
		,,* → {A1,A2,A3,A4}		
7.	12,5,5	h2,*,* → A4	A2	A2
	h1,p2,b1	h1,p1,b1 → A3	(from third connection)	
		h1,p2,b1 → A2		
		h3,p2,* → A1		
		h1,*,b2 → A4		
		,,* → {A1,A2,A3,A4}		

The actual situation descriptions, such as 4,28,3 in trial 2, are converted to abstract situation descriptions in a manner analogous to the parsing step of section 2.2. Thus 4,28,3 becomes h1-h2,p1,b1, meaning that 4 is a member of set h1 and set h2 , 28 is a member of set p1 , and 3 is a member of set b1 . In trial 5 the actual element

9 is a member of no set and is consequently represented by the abstract element p .

In the training trials just described no S-A connections were placed in the connection list by merely modifying definitions because no connection already in the list had the same form as the ones being added to the list. A connection in the list has the same form as one being added to the list only if (1) their A's are identical, (2) for each * in the S of the connection being added there is a corresponding * in the S of the connection already in the list, and (3) their corresponding abstract elements both use the same logical operator. For example, consider the following S-A connections.

(a)	h1,*,b1 → A1		h1: H < 12
(b)	h1,*,b2 → A1		h2: H < 6
		where	b1: B > 7
(c)	h1,*,* → A2		b2: B < 15
(d)	h2,*,b3 → A1		b3: B > 2

Here (a) and (b) are not of the same form because of restriction (3), (a) and (c) are not of the same form because of restriction (1), and (a) and (d) are of the same form.

The process of placing a connection in the list by modifying definitions is described below for the learning of the connection "18,24,3 → A3 because 18 is small, 24 is large, and 3 is small".

S	Learned S-A Connections	Predicted A	Correct A
18,24,3	h2,*,* → A4	A1	A3
= h,p1,b1	h1,p1,b1 → A3	(from last connection)	
	h1,p2,b1 → A2		

S	Learned S-A Connections	Predicted A	Correct A
	h3,p2,* → A1		
	h1,*,b2 → A4		
	,,* → {A1,A2,A3,A4}		

It is assumed that the wrong action was predicted using the last connection is the above list, hence the model must add the connection h4,p1,b1 → A3 to the list. Here h4 is defined by the set "H < 19", and this is learned when the model is told that 18 is "small". The model searches all connections above the error-causing one to see if any have the same form as h4,p1,b1 → A3. In the above example, only the second connection, h1,p1,b1 → A3, has this form. Consequently, the definition of h1 is changed to include 18, thus its new definition is h1: H < 19. Now when 18,24,3 is given to the model it predicts the correct action, A3.

EVALUATION OF THE MODEL. The Complex-placement model, like the Induction model, offers a more satisfying explanation of human cognition than do the first two models described. Also, for this model, the response retrieval time does not sharply increase as the number of reinforced training trials increases. This is because (a) the retrieval process does not always require looking at every connection in the list, and (b) a new connection is not always added to the connection list when an error is made. Moreover, the Complex-placement model does not require consistent training trials, as does the Induction model. If the model is told that h1,p1,* → A1 is a valid connection, and then that h1,*,b2 → A2 is a valid connection, it has been given inconsistent information, since in situation h1,p1,b2 two different actions should

be taken. Nonetheless, this information is incorporated into the ordered connection list. If the second connection is placed in the list because the first connection led to an error, the list has the following form:

h1,*,b2 → A2
h1,p1,* → A1
,,* → {A1,A2}

But now because of the hierarchical arrangement of the connections in the list the information is no longer inconsistent. The list in effect says to take action A1 if H is h1, P is p1 and B is anything but b2, and to take action A2 if H is h1, P is anything, and B is b2.

The Complex-placement model does, however, have at least one shortcoming. In the early stages of training it often resorts to making predictions at random, since it is difficult to find an exact match when the connection list is short. This might have a detrimental effect on the degree of correlation obtainable between the predictions made by the model and the predictions made by human subjects.

4.3. PROPOSED EXPERIMENTAL DESIGNS

In the previous section a number of S-R theories or models of human strategy learning were presented. The validity of these models can be tested by comparing them with human subjects in a game-playing or problem-solving environment.

Random Selection Design

An experimental paradigm for testing these models is outlined below. It is patterned after a series of experiments performed by Hunt, Marin, and Stone (1966) which are based on a random selection design.

1. Choose a game-playing or problem-solving environment. For this environment define (a) a set of attributes with numerical values, such that a situation description consists of a list of the values of these attributes, (b) a set of actions which can be taken, and (c) a set of consistent strategies in the form of situation-action pairs with exclusive abstract values, which partitions the universe of possible situations into n subsets, one for each possible action.
2. Pick a group of situation descriptions at random from the universe of possible situations.
3. Present these situation descriptions to the subjects in a serial fashion, and for each presentation or trial ask the subjects to predict the correct action. After each subject makes a prediction give him the correct action, and the reason why the action is correct, expressed as an evaluation of the relevant attributes. Present this information visually, such that on subsequent trials the subject

has available a cumulative visual record of the results of all previous trials.

4. Compare the predictions of the models with the predictions of the human subjects, when the models are given the situation descriptions from step 2, presented in the same order as they were presented to the subjects.

TRAINING INFORMATION. The information given to the subjects after each prediction can be obtained in a variety of ways. One way is to separately analyze each situation description from step 2 and decide, on the basis of the particular environment being represented, what action should be taken and why. The danger here is the possibility of inadvertently giving the subjects inconsistent information.

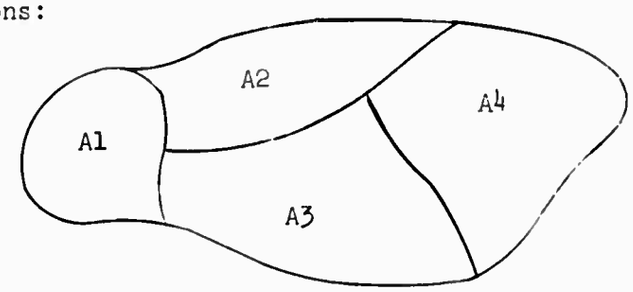
A better way to obtain the desired information is to use the set of S-A pairs defined in step 1 to grow a decision tree, using the generalization technique described for the Induction model. Each situation description, S , of step 2 is then sorted down the tree, and the correct action is assumed to be the one contained in the terminal node reached by S . As this S is sorted down the tree it passes through a number of test nodes which define its path through the tree. All attributes which are tested by these path-defining nodes are considered to be attributes relevant to choosing the correct action for S . The evaluation of these relevant attributes (or the reason why the action is taken) is simply the specification of the categories they fall into. The available categories are those defined by the exclusive definitions used to specify the abstract values needed for the set of S-A pairs defined in step 1.

TRAINING TRIALS. The training trials used in section 4.2 to describe the operation of the models were obtained by the method just outlined. The environment chosen is shown in Figure 4-3, and the tree grown from the S-A pairs in Figure 4-3 is shown in Figure A-1. To see how the training trials were constructed, consider the situation description 12,9,10 used in the training sequence of Figure 4-2. This description becomes h2,p2,b1 when expressed in terms of the abstract values defined in Figure 4-3, thus h2,p2,b1 is sorted down the tree of Figure A-1. The terminal node reached contains A4 , so the correct action is assumed to be A4 . The path that h2,p2,b1 takes through the tree is defined by the test nodes (h1?) (h2?) and (b1?) , thus attributes H and B are assumed relevant. The reason A4 is correct is therefore because H is an h2 , and B is a b1 . A game-playing interpretation of the environment defined in Figure 4-3 is presented in Appendix A, part II.

Rather than giving the subjects nondescriptive category names like h1 , h2 , and h3 they are given descriptive names which suggest how to order the categories, like large, medium, and small. Thus for trial 1 in Figure 4-2 the correct action is A3 because "H is medium, P is large, and B is small". If the models are to be compared to human subjects they must be given the training information in the same form used for the subjects. Consequently, the Induction model and the Complex-placement model (the models which learn the definitions of the abstract values) are given ordering information about the categories used to describe the attribute values, e.g., that "large" > "medium" > "small".

Attributes:	H	P	B
Range of Values:	1-50	1-60	1-10
Abstract Values:	h1(H>25)	p1(P>9)	b1(B>7)
	h2(10<H≤25)	p2(P≤9)	b2(B≤7)
	h3(H≤10)		

Universe of Situations:



Universe consists of 50x60x10 or 30,000 situations

Heuristics:

h1,*,b2 → A1
h1,p2,* → A1
h2,p2,b2 → A2
h1,p1,b1 → A3
h2,p1,b2 → A3
h2,*,b1 → A4
h3,*,* → A4

Figure 4-3. An environment for testing models of human strategy learning.

The Induction model can then use the ordering information to translate "large" into h1 , "medium" into h2 , and "small" into h3 when it is told why a particular action is correct, and then proceed as described in section 4.2. The Complex-placement model must use the ordering information to translate any given category into either "large" or "small". It can accomplish this by interpreting all categories above the middle one as "large", all below the middle one as "small", and the middle one itself (if there is one) as "small". Thus it would interpret "large", "medium", and "small" as "large", "small", and "small" when told why a particular action is correct, and proceed as described in section 4.2.

Interactive Selection Design

Another experimental design which might prove interesting is one where interactive selection (Hunt, Marin, and Stone, 1966) is used in step 2 rather than random selection. Here the subject examines the entire universe of situation descriptions and decides for himself which situation description to consider for each trial. The models must likewise decide which situation description to pick for each trial, and an S should be picked which provides a good test of the training information received when the last error was made.

For the Induction model this requirement is satisfied if it is required to pick an S that sorts to the same terminal node as the S part of the S-A connection last added to the list from which the tree was grown. The requirement is satisfied for the Complex-placement model if it is required to pick an S which catches on or below the last S-A connection added to the list. It is difficult to satisfy this requirement

for the Simple and Stimulus Generalization models, consequently, they would not be included in an experiment based on interactive selection.

CHAPTER 5

A SPECIFIC APPLICATION

5.1 INTRODUCTION

In order to demonstrate the feasibility of the representation and manipulation techniques presented in chapters 2 and 3 a full scale application in the area of game playing will now be described. The game chosen for this task is basic draw poker, a game in which the players do not have access to all the existing game information. In contrast, games like chess, checkers, go, and backgammon are designed so that each player has available the total game information at each decision point; these are called games of perfect information (Rapport, 1966).

To date, research in heuristic game playing has been concerned predominately with games of perfect information, because these games can usually be represented by game trees in which very effective search and prediction procedures (such as minimaxing) are applicable. Minimaxing cannot be used with most games of imperfect information, as there is not enough information available to construct a game tree in advance. The representation and manipulation techniques described earlier are an effective approach to implementing decision-making and learning in an imperfect information environment.

Game playing is studied not merely to develop programs which are good at playing games, but more to develop programmable methods and techniques for solving practical problems. Games of imperfect information are useful to study because they are realistic abstractions of the complex

BLANK PAGE

problems encountered in daily life, moreso than games of perfect information. For example, chess is actually a game of war, where each side tries to defeat the other by capturing the opposing army and imprisoning the king. In actual war it is seldom the case that one side knows the exact location, strength, and capabilities of all units of the opposing army, as one does in the game of chess.

A similar analogy can be drawn between games of imperfect information and the struggle which occurs between businesses engaged in marketing competitive products. Again, each side is faced with the problem of making crucial decisions without having available the information needed for accurately predicting what the counter-move by the opposition will be.

In this chapter a detailed analysis of the heuristics for the bet decision in draw poker will be presented together with their representation as production rules and an illustration of their use in an actual computer program. Next, the process of training will be illustrated by showing how the program can be trained to play draw poker, using either a human or a program as a trainer. Finally, it will be shown how the program can learn to play poker without explicit training, that is, by gaining experience through actual game play.

5.2 HEURISTICS FOR DRAW POKER

The game under consideration is a standard version of five-card draw poker, in which up to three cards may be replaced and no cards are wild. (See Appendix B, Part I for a detailed definition of the game.) The bet decision made by the computer program which plays this game is based on a number of interrelated heuristics. An informal description of these heuristics is given in Appendix B, Part II.

State Vector Description

The state vector needed to adequately describe the bet decision heuristics for this game has the form:

$$\mathcal{E} = (\text{VDHAND},\text{POT},\text{LASTBET},\text{BLUFFO},\text{POTBET},\text{ORP},\text{OSTYLE},\text{OH},\text{OB},\text{CS},\text{BO},\text{LAP},\text{SB},\text{MB},\text{BB},\text{BBS},\text{BBL},\text{OAVGBET},\text{OTBET},\text{OBLUFFS},\text{OCORREL},\text{OD}) ,$$

where the dynamic variables are VDHAND, POT, LASTBET, BLUFFO, POTBET, ORP, and OSTYLE , the function variables are OH, OB, CS, BO, LAP, SB, MB, BB, BBS, and BBL , and the bookkeeping variables are OAVGBET, CTBET, OBLUFFS, OCORREL, and OD . The definitions of these variables and the definitions of the symbolic values of variable VDHAND are presented in Figure 5-1.

The range of values for BLUFFO, OSTYLE, OH, OB, CS, BO, and OCORREL is the set of positive and negative integers, where a large or positive value indicates a high probability that the opponent can be bluffed, the opponent is conservative, etc. VDHAND ranges from 1 for one-of-a-kind to 600,000 for a royal flush, LASTBET ranges from 1 to 20 , and ORP ranges from 0 to 3 . VDHAND is an exclusive variable, while the other dynamic variables are of the overlapping type. It should be noted that

in two instances a variable serves a dual role, being both a function and a dynamic variable, i.e., BO and BLUFFO both stand for the same variable, and CS and OSTYLE both stand for the same variable.

The subvector for this game is composed of the dynamic variables of the state vector and thus has the form:

$$\beta = (\text{VDHAND},\text{POT},\text{LASTBET},\text{BLUFFO},\text{POTBET},\text{ORP},\text{OSTYLE}) .$$

For convenience the dynamic variables will be abbreviated so that the subvector can be written:

$$\beta = (H, P, B, BFO, PB, R, OCS) .$$

VDHAND:	the value of your hand
PCT:	the amount of money in the pot
LASTBET:	the amount of money last bet
BLUFFO:	a measure of the probability that the opponent can be bluffed
POTBET:	the ratio of the money in the pot to the amount last bet
ORP:	the number of cards replaced by the opponent
OSTYLE:	a measure of conservative style by the opponent
OH:	the expected value of the opponent's hand
OB:	a measure of the probability that the opponent is bluffing
CS:	a measure of conservative style by the opponent
BO:	a measure of the probability that the opponent can be bluffed
LAP:	the largest bet possible without causing the opponent to drop
SB:	a small bet
MB:	a medium size bet
BB:	a large bet made in an attempt to bluff the opponent
BBS:	a small bluff bet
BBL:	a large bluff bet
OAVGBET:	the average bet made during a round of play
OTBET:	the number of bets made by the opponent during a round of play
OBLUFFS:	the number of times the opponent was caught bluffing
OCORREL:	a measure of the correlation between the opponent's hands and bets
OD:	the number of times the opponent has dropped
SW:	a sure-to-win hand
EC:	an excellent-chance-of-winning hand
GC:	a good-chance-of-winning hand
PC:	a poor-chance-of-winning hand
NC:	a no-chance-of-winning hand
K1 to K31:	constants

Figure 5-1. Definitions of state vector variables and symbolic values.

The Heuristics As Production Rules

The bet decision heuristics (described in Appendix B, Part II) by virtue of being informal are also imprecise and occasionally ambiguous. However, they can be made precise and unambiguous by being rewritten and expanded in LASH, a language designed for specifying heuristics (see section 2.3). The LASH version of the bet decision heuristics are given in Appendix B, Part III, and the corresponding production rules in Appendix B, Part IV.

The five function variables OH,OB,CS,BO, and LAP are highly interrelated as can be seen from ff rules 11 through 14 in Appendix B, Part IV. The relationships existing between these variables and the bookkeeping variables are illustrated in Figure 5-2. OAVGBET and OTBET can be thought of as contributing to the short-term memory of the system while OBLUFFS,OCORREL and OD contribute to the long-term memory. Extending this idea, VDHAND,POT,LASTBET,POTBET, and ORP are short-term variables while BLUFFO and OSTYLE are long-term variables. The value of the constants used in defining these variables are given in Appendix B, Part V.

The production rules representing the bet decision heuristics have been incorporated into a LISP (McCarthy, 1962) computer program which plays draw poker. A listing of the action rules and bf rules actually used by the program is shown in Figure 5-3. The expression (INCP) in the action rules stands for the expression POT+(2 x LASTBET) . For each action rule in Figure 5-3 the first item in the rule is the left part of that rule, with the last 7 items forming the right part of the rule.

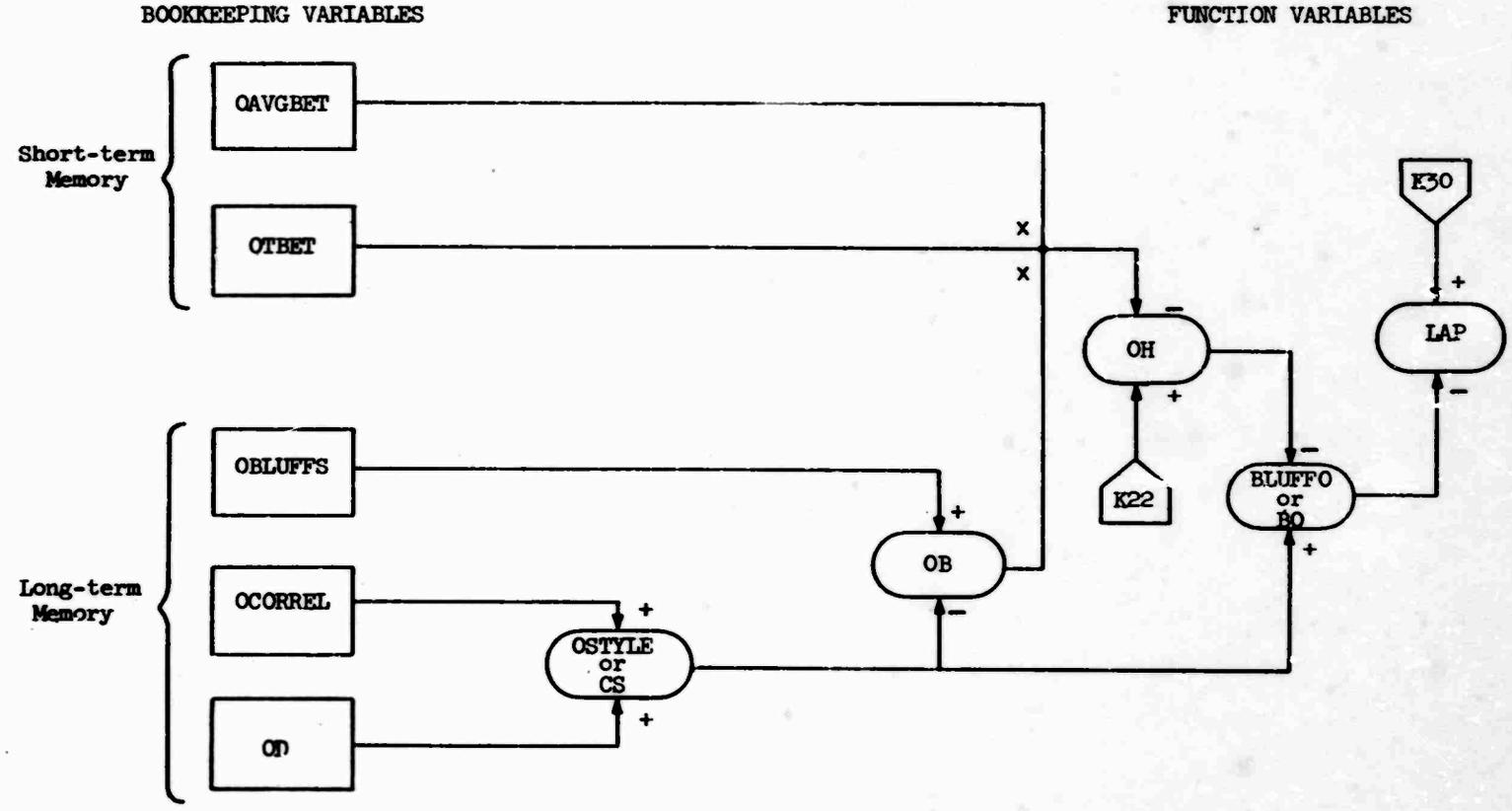


Figure 5-2. The relationships existing between the function variables and the bookkeeping variables.

```
(DEFPROP BUILT-IN-HEURISTICS
(NIL
 (((SW P8 B5 * * * *) * (INCP) 0 * * * *)
  ((SW * * * * * *) * (INCP) LAP * * * *)
  ((EC P1 B5 * * * *) * (INCP) 0 * * * *)
  ((EC * * * * * *) * (INCP) LAP * * * *)
  ((GC P2 B5 * * OR1 *) * (INCP) 0 * * * *)
  ((GC P9 B6 * * OR1 *) * (INCP) 0 * * * *)
  ((GC * B5 * * OR2 CS1) * (INCP) 0 * * * *)
  ((GC P3 B5 * * OR3 *) * (INCP) 0 * * * *)
  ((GC * * B01 * OR3 *) * (INCP) SU * * * *)
  ((GC P4 B5 * * * *) * (INCP) 0 * * * *)
  ((GC P9 B7 * * * *) * (INCP) 0 * * * *)
  ((GC * * * * * *) * (INCP; MH * * * *)
  ((PC * B5 * PB2 OR4 *) * (INCP) 0 * * * *)
  ((PC * B5 * PB2 OR2 CS2) * (INCP) 0 * * * *)
  ((PC P6 B9 B01 PB3 OR6 *) * (INCP) MH * * * *)
  ((PC P5 B2 B02 * * *) * (INCP) BB * * * *)
  ((PC * B8 * PB4 OR6 *) 0 * 0 * * * *)
  ((PC * B5 * * * *) * (INCP) 0 * * * *)
  ((PC * * * * * *) * (INCP; SB * * * *)
  ((NC * * * * OR4 *) 0 * 0 * * * *)
  ((NC * * * * OR2 CS3) 0 * 0 * * * *)
  ((NC P10 B9 B01 * OR7 *) * (INCP) B8S * * * *)
  ((NC P6 B4 B03 * OR6 *) * (INCP) 8BL * * * *)
  ((NC * B5 * PB1 * *) * (INCP) 0 * * * *)
  ((NC P7 B9 * * * *) * (INCP) 0 * * * *)
  ((NC P7 B3 * * * *) * (INCP) SH * * * *)
  ((NC P6 B3 * * OR6 *) * (INCP) SB * * * *)
  ((NC * * * * * * *) 0 * 0 * * * *))
 (((SW AND (GREATERP (DIFFERENCE H OH) K18) (NOT (LESSP H K19)))
   (EC AND (GREATERP (DIFFERENCE H OH) K18) (LESSP H K19))
   (GC AND
       (LESSP K20 (DIFFERENCE H OH))
       (NOT (GREATERP (DIFFERENCE H OH) K18)))
   (PC AND
       (LESSP K21 (DIFFERENCE H OH))
       (NOT (GREATERP (DIFFERENCE H OH) K20)))
   (NC NOT (GREATERP (DIFFERENCE H OH) K21)))
  ((P1 GREATERP P K1) (P2 GREATERP P K2)
                      (P3 GREATERP P K4)
                      (P4 GREATERP P K6)
                      (P5 LESSP P K9)
                      (P6 LESSP P K14)
                      (P7 LESSP P K32)
                      (P8 GREATERP P 120)
                      (P9 GREATERP P 17)
                      (P10 LESSP P 15))
  ((B1 LESSP B K8) (B2 LESSP B K10)
                   (B3 LESSP B K13)
                   (B4 LESSP B K15)
                   (B5 GREATERP H 0)
                   (B6 GREATERP H 7)
                   (B7 GREATERP B 12)
                   (B8 GREATERP B 11)
                   (B9 AND (LESSP B 5) (NOT (EQUAL B 0))))
  ((B01 GREATERP BFO K5) (B02 GREATERP BFO K11)
                         (B03 GREATERP BFO K16))
  ((PB1 GREATERP PB K17) (PB2 GREATERP PB 1)
                         (PB3 GREATERP PB 3)
                         (PB4 LESSP PB 2))
  ((OR1 OR (EQ R 0) (EQ R 1)) (OR2 EQUAL R 2)
                              (OR3 EQUAL R -1)
                              (OR4 EQUAL R 0)
                              (OR5 EQUAL R 1)
                              (OR6 NOT (EQUAL R -1))
                              (OR7 EQUAL R 3))
  ((CS1 GREATERP OCS K3) (CS2 GREATERP OCS K7)
                         (CS3 GREATERP OCS K12))))
VALUE)
```

Figure 5-3. Built-in heuristics for draw poker.

A Proficiency Test for Poker

In the next section it will be shown how training can produce useful and effective sets of heuristics. In order to test the poker playing ability of the programs which are trained, some type of proficiency test is needed. Such a test will now be described and applied to the poker program as it uses the heuristics (28 action rules and 41 bf rules) given in Figure 5-3 (henceforth referred to as the "built-in" heuristics). Applying this test to the program containing the built-in heuristics will provide a base against which the heuristics learned through training can be compared, in terms of game-playing effectiveness.

TEST PROCEDURE. The proficiency test consists of the following procedure. The program plays 5 games against a human opponent, each consisting of 5 hands. The cards are dealt from a standard deck of 52 cards which is first shuffled in a random manner. When the deck is depleated the cards are shuffled and the same deck is used again. Thus a total of 50 hands are dealt during the 5 games, 25 to the program and a corresponding 25 to the human opponent. (In this context a hand is taken to mean the 5 cards dealt plus 3 additional cards which may be given to the player if he decides to replace cards from his original five.)

After the 5 games are played a second series of 5 games is played, again using the same hands that were used in the first series. However, in the second series of games the program receives the 25 hands held by the opponent in the first series, and the opponent receives the 25 corresponding hands held by the program in the first series.

Series	Game	Program Hand	Opponent Hand	Series	Game	Program Hand	Opponent Hand
I.	1.	a	a'	II.	1.	u'	u
		b	b'			v'	v
		c	c'			w'	w
		d	d'			x'	x
		e	e'			y'	y
	2.	f	f'		2.	k'	k
		g	g'			l'	l
		h	h'			m'	m
		i	i'			n'	n
		j	j'			o'	o
	3.	k	k'		3.	f'	f
		l	l'			g'	g
		m	m'			h'	h
		n	n'			i'	i
		o	o'			j'	j
	4.	p	p'		4.	a'	a
		q	q'			b'	b
		r	r'			c'	c
		s	s'			d'	d
		t	t'			e'	e
	5.	u	u'		5.	p'	p
		v	v'			q'	q
		w	w'			r'	r
		x	x'			s'	s
		y	y'			t'	t

Figure 5-4. Possible arrangements of hands for the proficiency test for draw poker.

This procedure is illustrated in Figure 5-4. It is seen that in series I the program receives hands a through y , and the opponent hands a' through y' . In series II the situation is reversed; the program receives a' through y' and the opponent a through y . The only difference between series I and series II, other than the reversal of hands, is that the games do not occur in the same order. For example, in Figure 16, game 1 of series I occurs as the fourth game of series II. The games of series I are rearranged by a random process to establish the game order for series II.

PLAYING ABILITY. The playing ability of the program is measured relative to the opponent's playing ability as follows. The amount won by the program in series I is compared to the amount won by the opponent in series II for corresponding r-o-p's , and these results are displayed in graphical form as illustrated below.

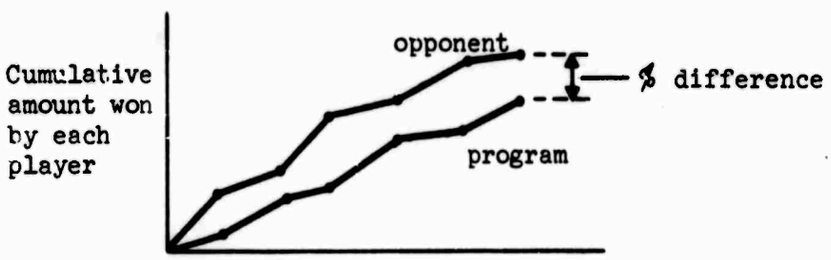


Figure 5-5.

Also calculated is the percentage difference between the total amount won by the opponent and the total amount won by the program. Since the same human opponent is used in each proficiency test, the test provides a means of comparing the game-playing effectiveness of different sets of heuristics.

In order to reduce the likelihood that the opponent remembers and uses information he is exposed to in series I as he plays the games of series II, (1) a number of dummy hands chosen randomly are played immediately before and after series I is played, and (2) a time elapse of 24 hours is used to separate series I from series II.

TEST RESULTS FOR BUILT-IN HEURISTICS. The results obtained by applying the proficiency test to the poker program containing the built-in heuristics are shown in Figure 5-6. It is seen that the program won roughly the same amount as the human opponent, who is an experienced player. In fact, the program won slightly more than the human opponent; i.e., the opponent won 5% less than the amount won by the program. A portion of the series of games which comprise this proficiency test is presented in Appendix C.

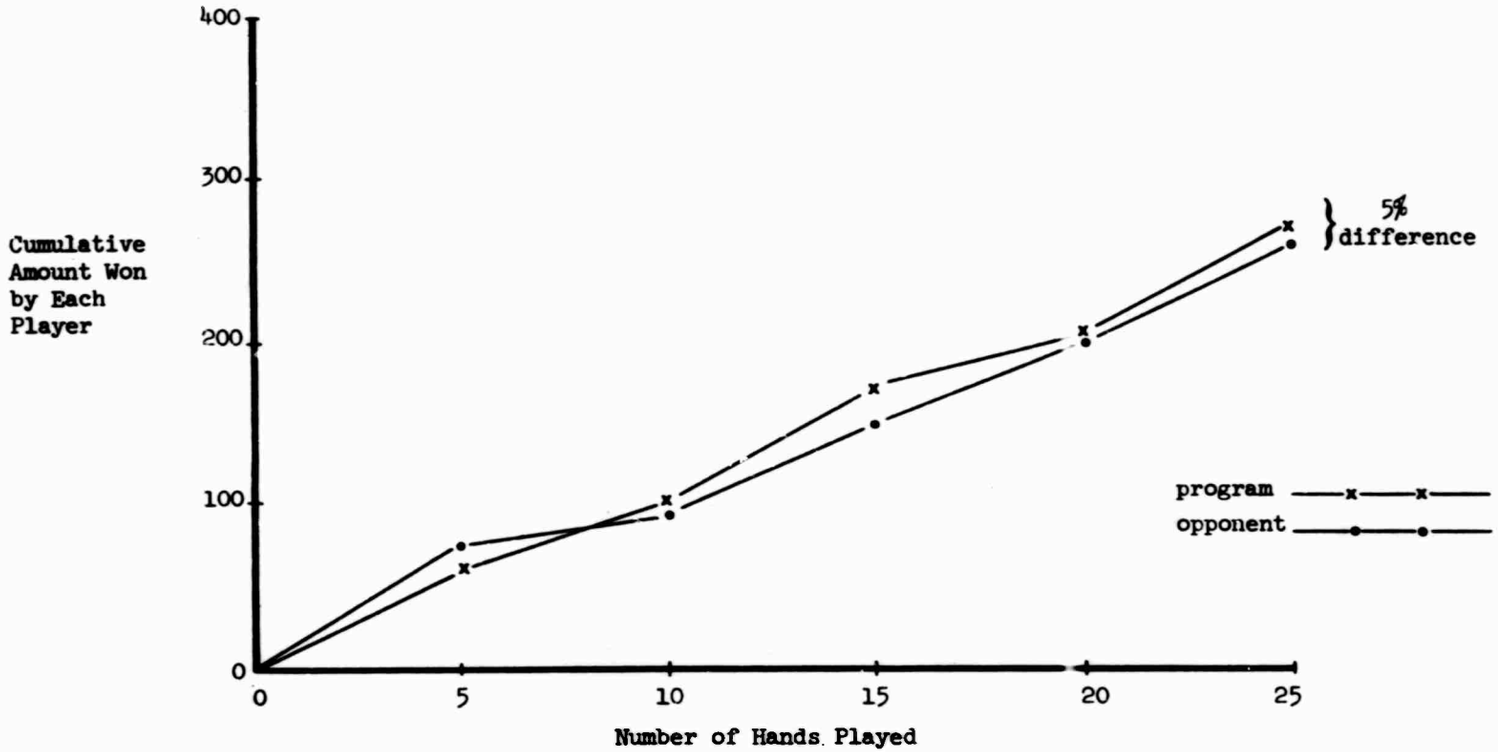


Figure 5-6. Results obtained by applying the proficiency test to the poker program containing the built-in heuristics.

5.3 TRAINING THE POKER PROGRAM

The training procedures described in section 3.2 will now be applied to the aforementioned system for playing draw poker. The program to be trained initially contains one action rule of the form

$$(*, *, *, *, *, *, *) \rightarrow (\text{random decision}) ,$$

no bf rules, and one ff rule for each of the function variables. During the course of training the program learns both the action rules and the bf rules, in a manner exactly identical to the process described earlier. In all examples discussed in this section training is continued to the point where further training results in little or no improvement in the program's ability to avoid making decisions which are rated unacceptable by the trainer.

Training Using a Human Trainer

In the first type of training to be illustrated the program plays an actual game against a human opponent and immediately after making each move decision asks a human trainer if the move was satisfactory. If the trainer indicates that the move was acceptable, the program proceeds by making that move. If the trainer instead indicates that a particular alternative move would have been better, the program analyzes the training information supplied by the trainer, incorporates it into the existing production rule list, and then proceeds by making the trainer-recommended move. This correction procedure is called a training trial. Thus a training trial occurs only when the program makes an error, that is, a decision which is unacceptable to the trainer.

The heuristics learned by the program after being put through 38 training trials by a human trainer are given in Figure 5-7. These heuristics will be referred to as the "manual-training" heuristics. During the training process 31 action rules were created, but 5 of these were made redundant through generalization on other rules and were automatically removed after training was completed, leaving the 26 action rules shown in Figure 5-7. A portion of the training trials used to create the manual-training heuristics is presented in Appendix D.

TEST RESULTS FOR MANUAL TRAINING. In order to test the game-playing effectiveness of the manual-training heuristics the proficiency test was applied to the poker program containing these heuristics (see Appendix E for a sample of the games played for this test) and the results plotted in Figure 5-8. As the graph shows, the program won almost as much as the opponent did, winning 6.8% less than the amount won by the opponent.

```
(DEFPROP MANUAL-TRAINING-HEURISTICS
(NIL
 (((H3 * B3 * * * *) * (INCP) 0 * * * *)
  ((H3 P1 * * * * *) * (INCP) SB * * * *)
  ((H3 P14 B2 B03 * * *) * (INCP) BBS * * * *)
  ((H3 * B2 * * * *) * (INCP) SB * * * *)
  ((H4 P1 B7 B02 * * *) * (INCP) BB * * * *)
  ((H4 * B2 * * * *) * (INCP) SB * * * *)
  ((H4 P1 B8 * * * *) * (INCP) 0 * * * *)
  ((H4 * * * PB4 * *) 0 * 0 * * * *)
  ((H4 P3 B4 * * * *) * (INCP) 0 * * * *)
  ((H4 P1 * * * R1 *) * (INCP) SB * * * *)
  ((H2 * * B03 * R1 *) * (INCP) SB * * * *)
  ((H2 P1 * B04 * * *) * (INCP) BB * * * *)
  ((H2 P1 B2 * * * *) * (INCP) SB * * * *)
  ((H2 P8 B4 * * * *) * (INCP) 0 * * * *)
  ((H2 P2 B1 * * * *) * (INCP) 0 * * * *)
  ((H2 * * * * * *) * (INCP) MB * * * *)
  ((H3 P4 B5 * * * *) * (INCP) MB * * * *)
  ((H4 * * * * R4 *) 0 * 0 * * * *)
  ((H1 P4 * * * * *) * (INCP) SB * * * *)
  ((H1 P13 * * * * *) * (INCP) MB * * * *)
  ((H1 P6 * * * * *) * (INCP) LAP * * * *)
  ((H1 P9 B4 * * * *) * (INCP) 0 * * * *)
  ((H1 P10 B3 * * * *) * (INCP) 0 * * * *)
  ((H1 * * * * * *) * (INCP) LAP * * * *)
  ((H3 P12 B9 * * * *) * (INCP) 0 * * * *)
  ((H3 * * * * * *) * (INCP) SB * * * *)
  ((* * * * * * *) (STARO) (STARI) (BETO) * * * *))
 (((H4 LESSP (DIFFERENCE H (EVAL1 (QUOTE OH))) 0)
   (H3 AND
       (NOT (LESSP (DIFFERENCE H (EVAL1 (QUOTE OH))) 0))
       (LESSP (DIFFERENCE H (EVAL1 (QUOTE OH))) 12))
   (H2 AND
       (NOT (LESSP (DIFFERENCE H (EVAL1 (QUOTE OH))) 12))
       (LESSP (DIFFERENCE H (EVAL1 (QUOTE OH))) 34))
   (H1 NOT (LESSP (DIFFERENCE H (EVAL1 (QUOTE OH))) 34)))
  ((P1 LESSP P 3) (P2 GREATERP P 17)
                  (P3 GREATERP P 1)
                  (P4 LESSP P 13)
                  (P6 LESSP P 33)
                  (P8 GREATERP P 41)
                  (P9 GREATERP P 143)
                  (P10 GREATERP P 75)
                  (P12 GREATERP P 15)
                  (P13 LESSP P 23)
                  (P14 LESSP P 7))
  ((B9 NOT (EQUAL B 0)) (B8 AND (NOT (EQUAL B 0)) (LESSP B 4))
                        (B1 GREATERP B 4)
                        (B2 LESSP B 1)
                        (B3 GREATERP B 3)
                        (B4 GREATERP B 1)
                        (B5 LESSP B 2)
                        (B7 LESSP B 3))
  ((B02 GREATERP BFO 17) (B03 GREATERP BFO 0) (B04 LESSP BFO -5))
  ((PB4 LESSP PB 4))
  ((R4 EQUAL R 0) (R1 EQUAL R -1))
  NIL))
VALUE)
```

Figure 5-7. Manual-training heuristics for draw poker.

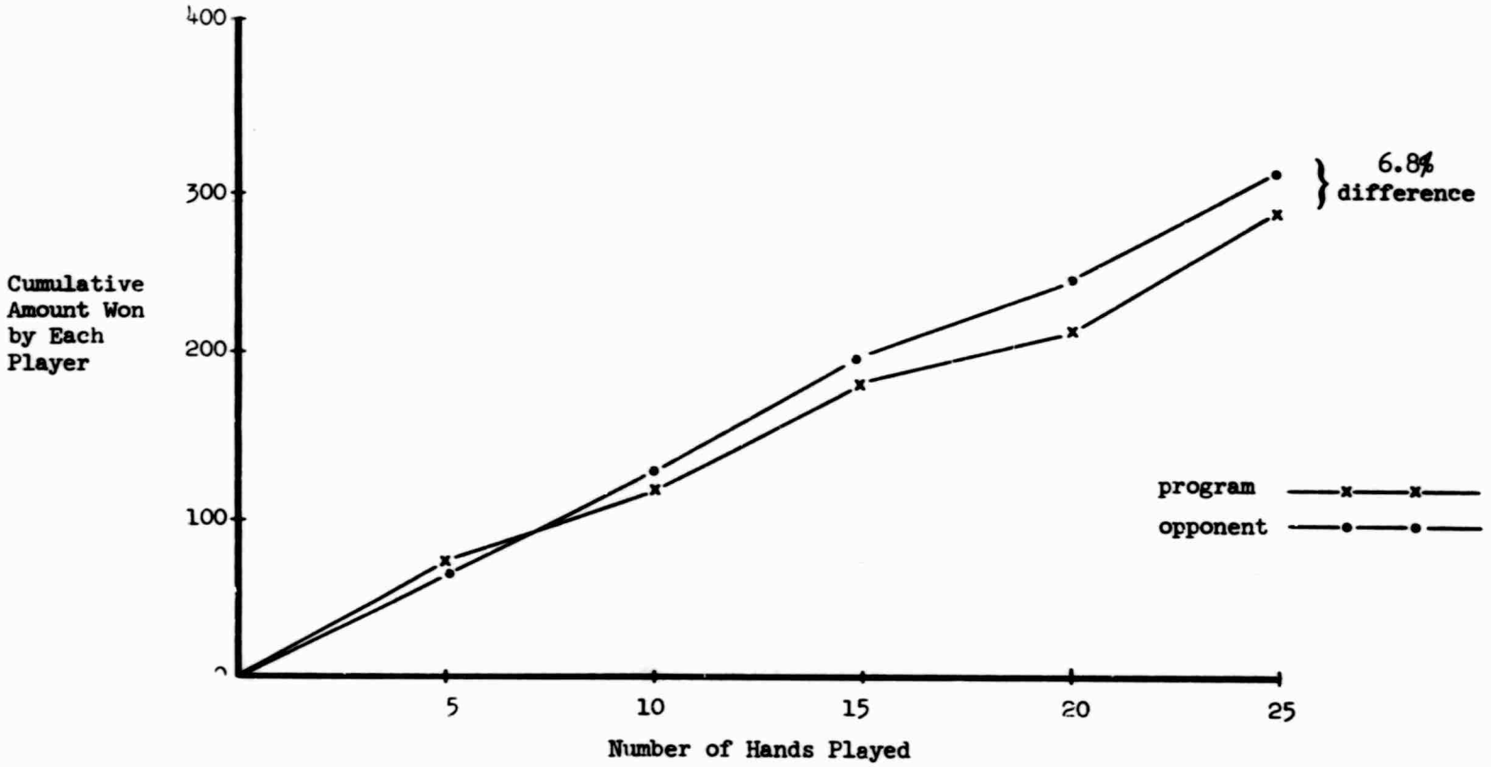


Figure 5-8. Results of applying the proficiency test to the poker program containing the manual-training heuristics.

Comparing this with the performance of the program containing the built-in heuristics it appears that although both programs play roughly as well as the human opponent the program with the built-in heuristics is somewhat superior to the program with the manual-training heuristics.

The improvement in game-playing ability due to training can be illustrated by comparing the results of the proficiency test applied before training (see Appendix F) with the results of the test applied after training. Figure 5-9 shows the results before training, where the program contained no bf rules and only one action rule of the form (* * * * * * *) → (random decision). Before training, as the graph shows, the program won 71% less than the amount won by the opponent, while after training it won only 6.8% less. Thus the training process effected a significant improvement in the playing ability of the program.

Training Using a Program Trainer

Training can also be implemented using a program rather than a human as the trainer. This method of training will now be illustrated, using the poker program containing the built-in heuristics as the trainer and another version of the poker program containing only the random decision action rule as the trainee. As before the trainee queries the trainer after each move decision to find if the move is acceptable. If it is not, the trainer supplies the trainee with the training information, in exactly the same form as that supplied by the human trainer, and the trainee incorporates it into its existing production rule list.

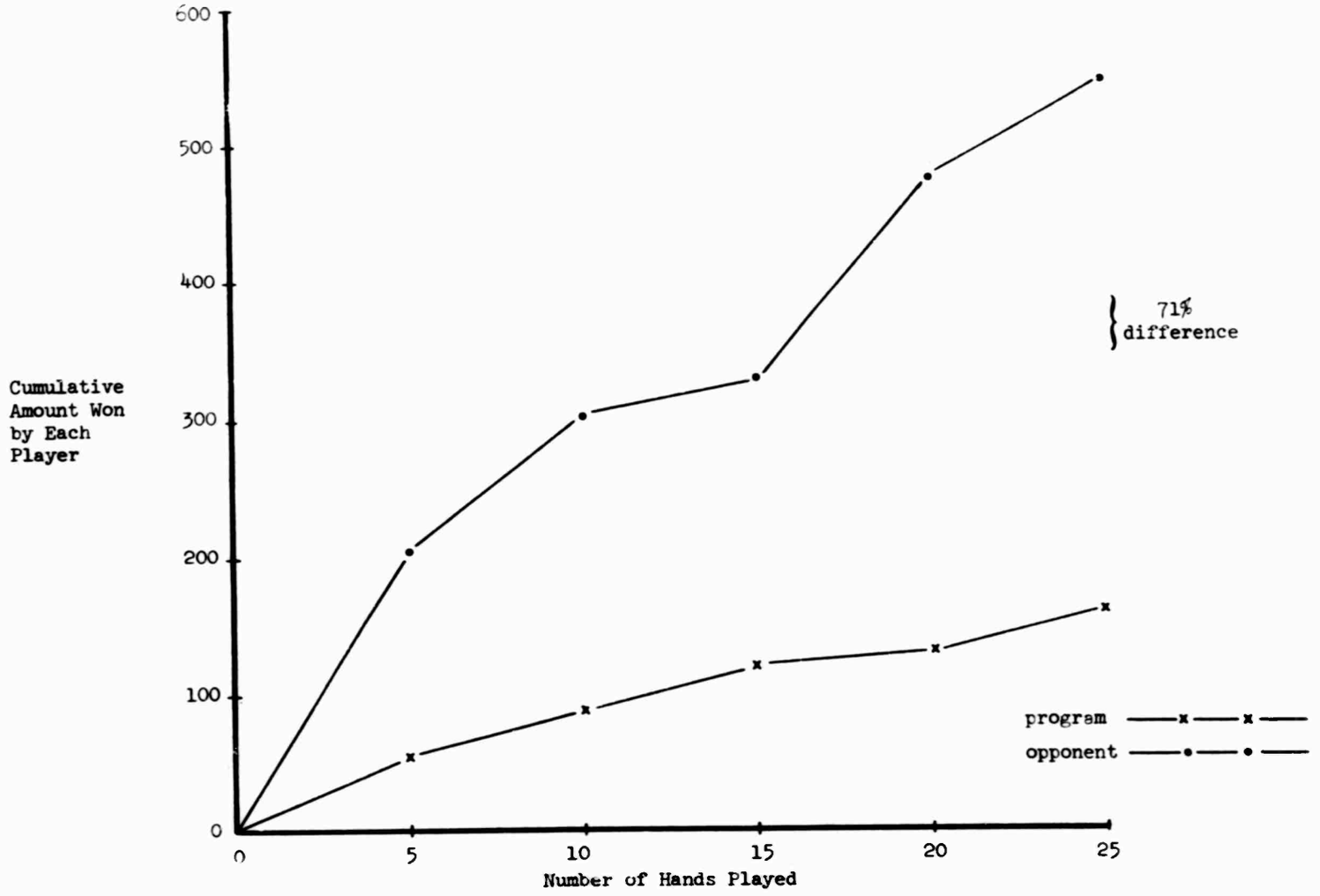


Figure 5-9. Results of applying the proficiency test to the poker program containing the before-training heuristics.

The effectiveness of the modification and generalization techniques used by the trainee as it learns how to play the game can be tested in the following manner. After training is completed the trainee plays a number of games against the human opponent and each decision made by the trainee is compared to the decision that the trainer would have made in that game situation. If the two programs rarely make the same decision it can be inferred that the modification techniques used by the trainee are ineffectual. On the other hand, if the trainer and trainee always make exactly the same decisions it can be inferred that the modification techniques used are extremely effective. In any case, the percentage of decisions which the two programs agree upon can be used as a measure of the effectiveness of the modification and generalization techniques.

A program trainer rather than a human trainer is used in obtaining this measurement because the program trainer by its very nature will make exactly the same decisions during testing as it did during the training process, whereas the human trainer cannot be relied upon to be this consistent. It should be clear that any inconsistency of this type exhibited by the trainer will decrease the percentage of decisions which the trainer and trainee agree upon, thus confounding the measurement of the effectiveness of the modification techniques.

The heuristics learned by the trainee after being put through 29 training trials by the program trainer are shown in Figure 5-10. These heuristics will be referred to as the "automatic-training" heuristics. During the training process 20 action rules were created, but one of these was made redundant through generalization on other rules and was

```
(DEFPROP AUTOMATIC-TRAINING-HEURISTICS
(NIL
 (((H3 * B3 * PB1 R2 *) 0 * 0 * * * *)
  ((H3 P1 B8 B04 * * *) * (INCP) BB * * * *)
  ((H3 P6 B4 B03 PB2 R2 *) * (INCP) BB * * * *)
  ((H3 P6 B10 B05 * * *) * (INCP) BB * * * *)
  ((H3 * B7 * * * *) * (INCP) 0 * * * *)
  ((H3 * * * * * *) * (INCP) SB * * * *)
  ((H2 * * B01 * R1 *) * (INCP) SB * * * *)
  ((H2 P4 B1 * * R3 *) * (INCP) 0 * * * *)
  ((H2 P5 B1 * * * *) * (INCP) 0 * * * *)
  ((H2 * * B03 * R1 *) * (INCP) SB * * * *)
  ((H2 P8 B6 * * R1 *) * (INCP) 0 * * * *)
  ((H2 * * * * * *) * (INCP) MB * * * *)
  ((H1 P3 B3 * * * *) * (INCP) 0 * * * *)
  ((H1 * * * * * *) * (INCP) LAP * * * *)
  ((H4 P2 B2 * * * *) * (INCP) SB * * * *)
  ((H4 P6 B4 * * * *) * (INCP) 0 * * * *)
  ((H4 P7 B2 * * R2 *) * (INCP) SB * * * *)
  ((H4 * B7 * PB3 * *) * (INCP) 0 * * * *)
  ((H4 * * * * * *) 0 * 0 * * * *)
  ((* * * * * * *) (STAR0) (STAR1) (BET0) * * * *))
 (((H4 LESSP (DIFFERENCE H (EVAL1 (QUOTE OH))) 0)
   (H3 AND
       (NOT (LESSP (DIFFERENCE H (EVAL1 (QUOTE OH))) 0))
       (LESSP (DIFFERENCE H (EVAL1 (QUOTE OH))) 13))
   (H2 AND
       (NOT (LESSP (DIFFERENCE H (EVAL1 (QUOTE OH))) 13))
       (LESSP (DIFFERENCE H (EVAL1 (QUOTE OH))) 34))
   (H1 NOT (LESSP (DIFFERENCE H (EVAL1 (QUOTE OH))) 34)))
  ((P1 LESSP P 11) (P2 LESSP P 3)
                   (P3 GREATERP P 63)
                   (P4 GREATERP P 43)
                   (P5 GREATERP P 47)
                   (P6 LESSP P 5)
                   (P7 LESSP P 15)
                   (P8 GREATERP P 13))
  ((B4 AND (LESSP B 5) (NOT (EQUAL B 0))) (B1 GREATERP B 2)
                                          (B2 LESSP B 1)
                                          (B3 GREATERP B 13)
                                          (B6 GREATERP B 1)
                                          (B7 GREATERP B 0)
                                          (B8 LESSP B 3)
                                          (B10 LESSP B 4))
  ((B01 GREATERP BF0 21) (B03 GREATERP BF0 10)
                         (B04 GREATERP BF0 27)
                         (B05 GREATERP BF0 46))
  ((PB1 LESSP PB 2) (PB2 GREATERP PB 3) (PB3 GREATERP PB 11))
  ((R3 OR (EQ R 0) (EQ R 1)) (R2 NOT (EQUAL R -1)) (R1 EQUAL R -1))
  NIL))
VALUE)
```

Figure 5-10. Automatic-training heuristics for draw poker.

automatically removed after training was completed, leaving the 19 action rules shown in Figure 5-10. A portion of the training trials used to create the automatic-training heuristics is given in Appendix G.

TEST RESULTS FOR AUTOMATIC TRAINING. The percentage of decisions which the trainer and the trainee agreed upon was measured, both before and after training, for 50 consecutive game situations supplied from hands chosen at random. The results are shown in Table 5-1 below.

% AGREEMENT BEFORE TRAINING	20%
% AGREEMENT AFTER TRAINING	96%

Table 5-1. Percentage agreement between trainer and trainee.

It is seen that training produces close to 100% agreement between the trainee and the trainer, thus showing that the modification and generalization techniques used are extremely effective.

The playing ability of the trainee, the poker program containing the automatic-training heuristics, was tested by applying the proficiency test to the program (see Appendix H for a sample of the games played). The results are plotted in Figure 5-11. As the graph shows, the program won approximately the same amount as did the opponent. Comparing Figure 5-11 with Figure 5-6 it appears that the trainee plays almost as well as the trainer, in spite of the fact that the trainee contains only 19 action rules, 9 less than the trainer contains.

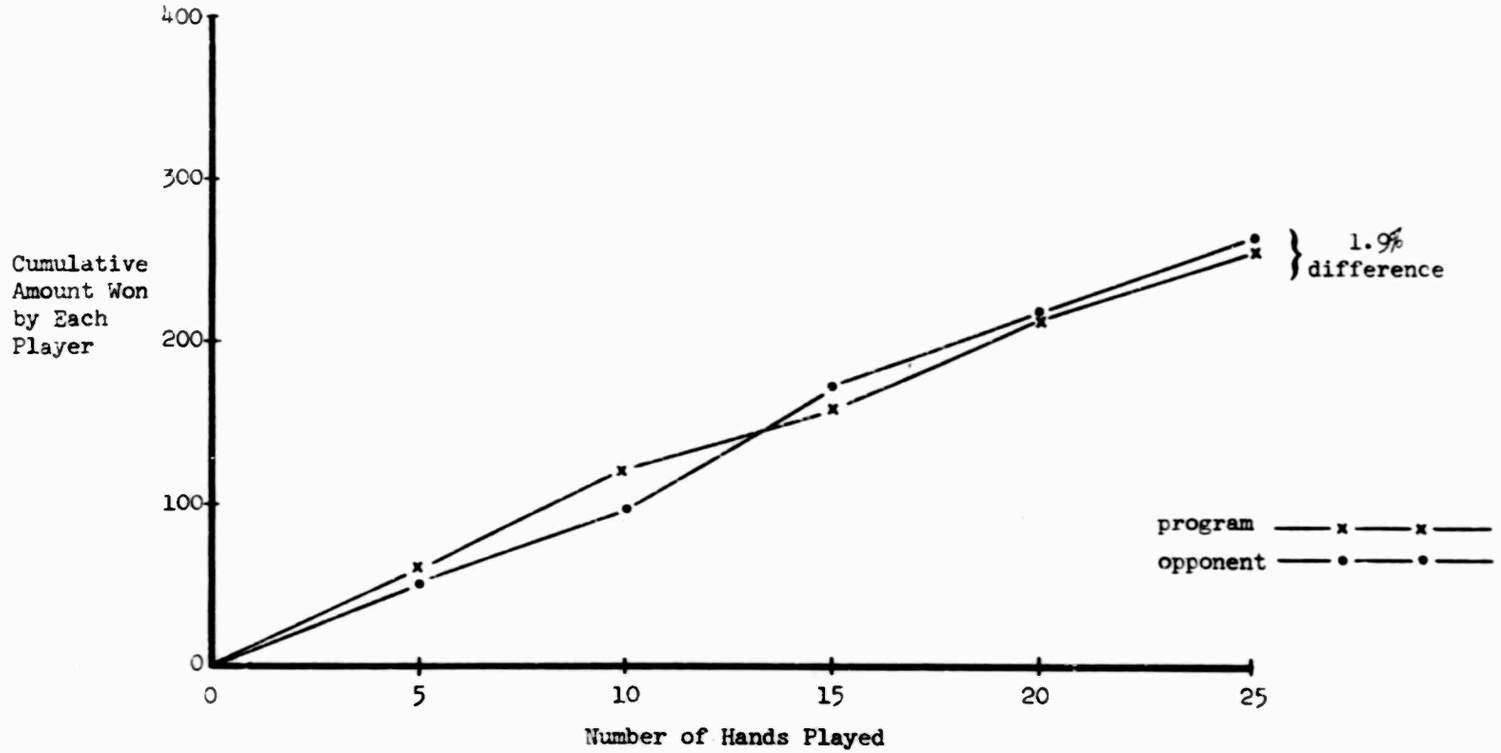


Figure 5-11. Results of applying the proficiency test to the poker program containing the automatic-training heuristics.

5.4 LEARNING POKER WITHOUT EXPLICIT TRAINING

The techniques described in section 3.3 which permit the program to obtain the training information through normal game play will now be applied to the problem of making the bet decision in draw poker. The program which uses this implicit-training procedure initially contains one action rule of the form (*, *, *, *, *, *, *) → (random decision), no bf rules, no ff rules, a set of logical statements or premises about the game of poker and game playing in general, and a decision matrix for poker. During the course of playing a series of games the program learns both the action rules and the bf rules.

Axiomatizing the Game

In order to permit the program to hypothesize reasonable heuristic rules without explicit training it is necessary to provide the program with a means of determining or deducing reasonable decisions. This can be accomplished by supplying the program with a set of logical statements based on

(1) the rules of the game,

(2) assertions (or "axioms") about the game,

(3) general propositions about techniques used in game playing.

Then, after the program makes a decision it can use these logical statements, together with information concerning the subsequent decision by the opponent and its effect on the game situation, to deduce what the original decision should have been.

PROGRAM OPERATION. Specifically, the program operates as follows. During a game the program subvector is saved each time a bet decision is made, and

this information is accumulated until the termination of the current round-of-play. If the r-o-p was terminated by a "drop", the information is not used; i.e., the program learns nothing. If the r-o-p was terminated by a "call", thus exposing the opponent's hand, a program subvector (and associated bet decision) is used, with the value of the opponent's hand, to set the predicates in the logical statements. Once these statements are so primed, the program is able to deduce what the bet decision should have been in order to have maximized the program's score. If the bet decision actually made by the program was not correct (the one that would have maximized the program's score) a learning trial takes place; i.e., the correct decision plus information from the decision matrix is used by the program to modify the existing production rule list as specified in section 3.3. This procedure is carried out individually for each program subvector (and associated bet decision) accumulated after cards are replaced.

NON-EVALUATABLE ACTION RULES. A major problem encountered in using this learning technique is that all action rules which specify the action DROP are non-evaluatable. This is true because when a drop is made the r-o-p is terminated but the program is not permitted to see the opponent's hand. Without this information the logical statements cannot be primed, consequently there is no way to determine whether or not the decision to drop was a sound one. This becomes a problem when a bad or ineffectual action rule leading to drop is hypothesized by the system, because it is non-evaluatable and thus cannot be modified or removed.

The problem of the non-evaluatable action rule is solved in the following way. If during the learning trials the symbolic subvector

catches on a non-evaluatable action rule the decision specified by the rule is not made, instead an evaluatable one (in this case a CALL) is made. Then during the evaluation process the non-evaluatable decision (the drop) is compared to the decision deduced using the logical statements, and if the two decisions differ the existing production list is modified. After learning is completed the substitution of evaluatable decisions for non-evaluatable ones is discontinued.

LOGICAL STATEMENTS. The logical statements used by the program are shown in Appendix I, Part I. The poker "axioms" included therein are statements which can be deduced by a human strictly from the rules of the game and an elementary knowledge of casual laws. It is reasonable to give these statements to the program since a human about to play the game for the first time would have this information readily available, even though he knew nothing of the decision strategy to use for the game.

The logical statements used by the program have the form $P \supset Q$, meaning that if P is true then Q is also true. The expressions P and Q consist of predicates and the logical connectives $\wedge$ and $\vee$. The arguments of a predicate may be either constants, as in add(pot,yourscore) or variables, as in add(x,z), and these variables may take the value of any constant as long as the assignment is consistent within a logical statement.

DEDUCTION PROCESS. To illustrate how the program can use these logical statements to deduce the best decision; i.e., the decision that would have maximized its score, consider the following. First, the state vector

associated with one of the program's bet decisions and the value of the opponent's hand are used to set certain predicates in the logical statements. Then the program takes the expression maximize(yourscore) and tries to make it true. To accomplish this the program searches the right sides of the implication statements $P \supset Q$ looking for a Q which matches maximize(yourscore) or can be made to match it by substituting constants for free variables. After such a Q is found the program applies the same technique to the problem of making the left side or P of the $P \supset Q$ statement true by matching P or parts of P against the right sides of the implication statements. This process continues until all decisions which make maximize(yourscore) true are found. An example of this deduction procedure is presented in Appendix I, Part III.

In some situations more than one type of action by the program will make maximize(yourscore) true. When this is the case the program chooses one of these actions as follows. The left side of general axiom 2 has the form $a \vee b \vee c$. If expression a can be made true then an action is picked at random from the set of actions which makes a true. If a cannot be made true but b can, then an action is picked at random from the set which makes b true. If neither a nor b can be made true then an action is picked at random from the set of actions which makes c true.

The Decision Matrix

As explained in section 3.3 a decision matrix is needed to provide the program with the reasons why the subvector variables are relevant. After the program logically deduces a decision and hypothesizes which variables are relevant, it uses the decision matrix to determine why

each of the variables hypothesized as relevant are in fact relevant. The decision matrix used for draw poker is shown below. Each row stands for a game decision and each column for a subvector variable.

	VDHAND	POT	LASTBET	BLUFFO	POTBET	ORP	OSTYLE
DROP	"Category the current value of VDHAND belongs in"	large	large	small	small	"current value of ORP"	large
CALL	"Category the current value of VDHAND belongs in"	large	large	small	large	"current value of ORP"	large
BET LOW	"Category the current value of VDHAND belongs in"	small	small	program hand : large good program hand : small poor	large	"current value of ORP"	large
BET HIGH	"Category the current value of VDHAND belongs in"	small	program hand : large good program hand : small poor	program hand : small good program hand : large poor	program hand : small good program hand : large poor	"current value of ORP"	program hand : small good program hand : large poor

Figure 5-12.

For example, if the program determines that the decision should have been BET LOW and hypothesizes that VDHAND, POT, LASTBET, BLUFFO, POTBET, ORP, and OSTYLE are relevant then it uses the decision matrix to find that it should make the decision BET LOW because VDHAND falls into a particular category, POT is small, LASTBET is small, BLUFFO is large (if goodhand(you) = T) or small (if goodhand(you) = F) , POTBET is large, ORP is a particular value, and OSTYLE is large.

Learning Based on Implicit Training

The effectiveness of the implicit-training techniques used by the learning program can be tested as follows. After learning is complete the program plays a number of games against the opponent and each decision made by the program is compared to the decision that would have been deduced in that game situation using the axiom set. The percentage of decisions agreed upon can be used as a measure of the effectiveness of the hypothesis-formation and deduction techniques used by the learning program.

The heuristics learned by the program after 57 training trials are shown in Figure 5-13. These heuristics will be referred to as the "implicit-training" heuristics. During the training process 15 action rules were created, but one of these was made redundant through generalization on other rules and was automatically removed after learning was completed, leaving the 14 action rules shown in Figure 5-13. A portion of the training trials used to create the implicit-training heuristics is given in Appendix J.

Learning was terminated after 57 training trials since this was the number of trials needed to make the action rules general enough to catch the symbolic subvector the vast majority of the time. After 57 trials they caught the symbolic subvector 95% of the time, permitting the random rule at the bottom of the action rule list to catch the subvector only 5% of the time.

TEST RESULTS FOR IMPLICIT TRAINING. The percentage of decisions agreed upon by the program and the axiom set was measured for 50 consecutive

```
(DEFPROP IMPLICIT-TRAINING-HEURISTICS
(NIL
 (((H1 * * * PB5 * CS5) * (INCP) SSS (DUMMY *) * * *)
  ((H3 * B3 * * * CS2) * (INCP) 0 (DUMMY *) * * *)
  ((H4 P32 B5 BO34 PB27 * CS12) * (INCP) BBB (DUMMY *) * * *)
  ((H4 P14 B12 * * * *) 0 * 0 (DUMMY *) * * *)
  ((H3 P27 B22 * PB5 * *) * (INCP) SSS (DUMMY *) * * *)
  ((H2 * B19 * PB5 * CS2) * (INCP) SSS (DUMMY *) * * *)
  ((H4 * B22 * * * *) * (INCP) SSS (DUMMY *) * * *)
  ((H3 * B12 BO2 PB7 * CS1) 0 * 0 (DUMMY *) * * *)
  ((H2 * B8 * PB5 * CS1) * (INCP) 0 (DUMMY *) * * *)
  ((H1 P22 B4 BO2 PB4 * CS6) * (INCP) BBB (DUMMY *) * * *)
  ((H1 * B3 * * * CS2) * (INCP) 0 (DUMMY *) * * *)
  ((H2 P15 * BO14 * R1 CS7) * (INCP) BBB (DUMMY *) * * *)
  ((H3 P12 * * * R1 *) * (INCP) BBB (DUMMY *) * * *)
  ((H2 P20 B4 * PB17 * *) * (INCP) BBB (DUMMY *) * * *)
  ((* * * * * * *) (STARO) (STARI) (BETO) * * * *))
 (((H4 LESSP H 3) (H3 AND (NOT (LESSP H 3)) (LESSP H 20))
                  (H2 AND (NOT (LESSP H 20)) (LESSP H 42))
                  (H1 NOT (LESSP H 42)))
  ((P12 LESSP P 27) (P14 GREATERP P 5)
                    (P15 LESSP P 21)
                    (P20 LESSP P 61)
                    (P22 LESSP P 31)
                    (P27 LESSP P 33)
                    (P32 LESSP P 23))
  ((B3 GREATERP B 4) (B4 GREATERP B 1)
                     (B5 LESSP B 4)
                     (B8 GREATERP B 7)
                     (B12 GREATERP B 0)
                     (B19 LESSP B 14)
                     (B22 LESSP B 6))
  ((BO2 LESSP BFO -5) (BO14 LESSP BFO 6) (BO34 GREATERP BFO -52))
  ((PB4 LESSP PB 17) (PB5 GREATERP PB 1)
                     (PB7 LESSP PB 41)
                     (PB17 LESSP PB 21)
                     (PB27 GREATERP PB 6))
  ((R1 EQ R 3))
  ((CS1 GREATERP OCS -1) (CS2 GREATERP OCS -2)
                         (CS5 GREATERP OCS -1)
                         (CS6 LESSP OCS 1)
                         (CS7 LESSP OCS 3)
                         (CS12 GREATERP OCS -6))))
VALUE)
```

Figure 5-13. Implicit-training heuristics for draw poker.

game situations, both before and after the training trials. The results are shown in Table 5-2 below.

% agreement before training	24%
% agreement after training	82%

Table 5-2. Percentage agreement between learning program and axiom set.

It is seen that the training trials produce an 82% agreement between the program and the axiom set, an increase of 58% over the agreement before training, thus showing that the implicit-training techniques are effective in implementing learning. The percentage agreement between the program and the axiom set (82%) was less than the percentage agreement between the trainee and trainer (96%) described in section 5.3.

The playing ability of the program containing the implicit-training heuristics was tested by applying the proficiency test to the program (see Appendix K for a sample of the games played). The results are plotted in Figure 5-14. As the graph indicates, the program won 13% less than did the experienced human opponent, implying that the opponent is a slightly better player than the learning program.

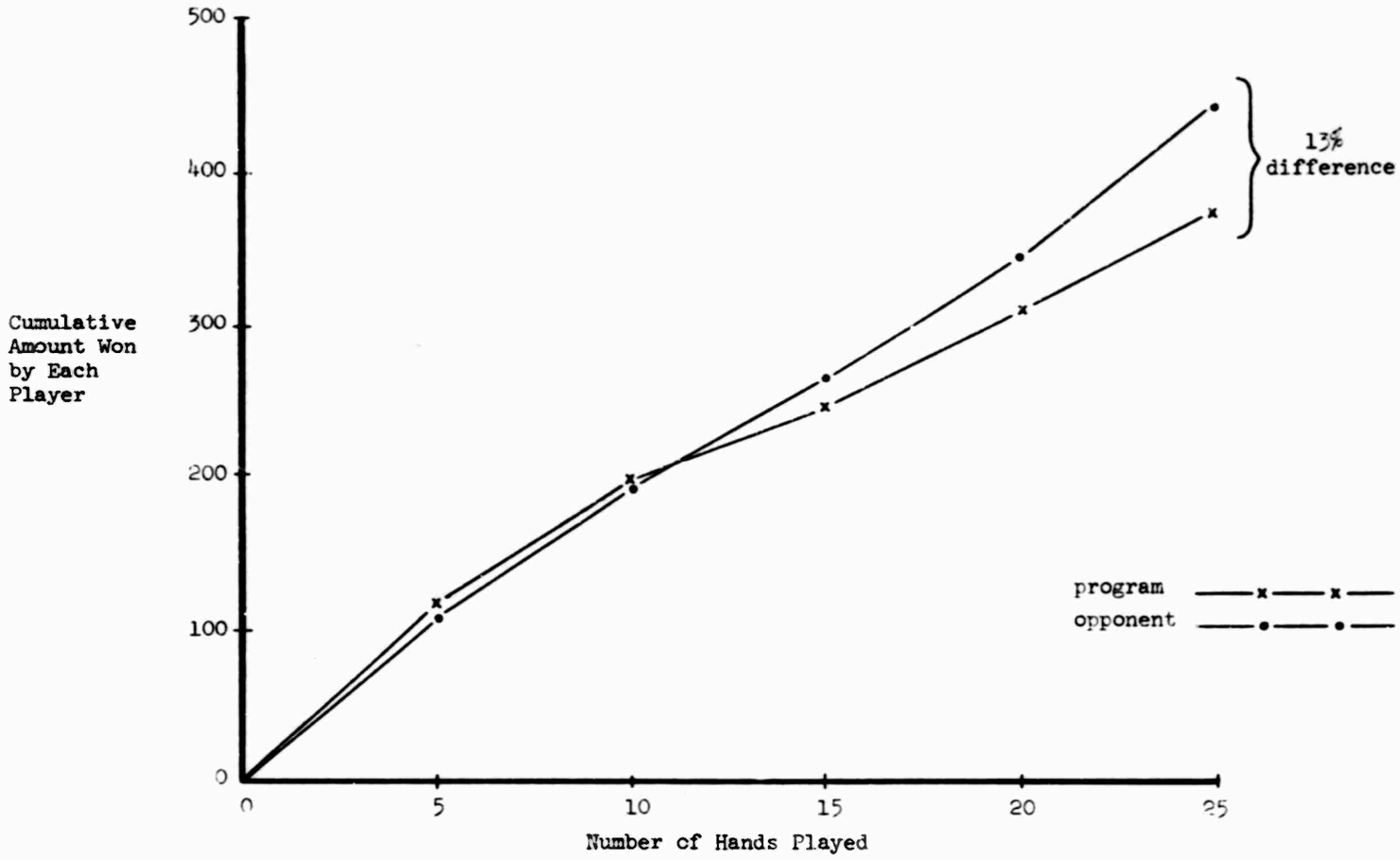


Figure 5-14. Results of applying the proficiency test to the poker program containing the implicit-training heuristics.

5.5 DISCUSSION OF RESULTS

The results obtained in sections 5.2, 5.3, and 5.4 are summarized in Table 5-3. The first column of this table is a list of the various sets of heuristics (action rules and associated bf rules) tested in this chapter. The before-training heuristics consist of a single action rule of the form (*, *, *, *, *, *, *) → (random decision) and no bf rules, whereas the other sets of heuristics consist of the action and bf rules illustrated in Figures 5-3, 5-7, 5-10, and 5-13.

NUMBER OF TRAINING TRIALS. The second column of Table 5-3 contains the number of training trials used to create the sets of heuristics listed in the first column of the table. The built-in and before-training heuristics were created by hand and thus required no training trials. The manual-training and automatic-training heuristics were created using the training procedure of section 3.2, and required 38 and 29 training trials, respectively. Training was continued until the trainee, during training, played one complete game of 5 hands without once making a decision rated unacceptable by the trainer. The implicit-training heuristics were created without the use of a trainer and required 57 training trials. Training was continued until the acquired action rules were made general enough to catch the symbolic subvector, and thus generate a non-random decision, 95% of the time.

The number of training trials required by the explicit-training procedures cannot be directly compared to the number of trials required by the implicit-training procedure because (1) the same criterion was not used in each case for determining when training trials should cease, and

Heuristics Used	Number of Training Trials	Number of Redundancies During Training	Number of Action Rules	Percent Difference in Winnings +: program ahead -: opponent ahead	Percent Agreement Between the Acting Trainer and the Trainee	
					Before Training Trials	After Training Trials
Built-in	-	-	28	+5%	-	-
Before-training	-	-	1	-71%	-	-
Manual-training	38	5	26	-6.8%	-	-
Automatic-training	29	1	19	-1.9%	20%	96%
Implicit-training	57	1	14	-13%	24%	82%

Table 5-3. Summary of Results.

(2) the number of decisions which had to be learned was not a constant, i.e., the explicit-training programs had to learn to associate 8 different decisions with the game situations encountered, while the implicit-training program had to do the same with only 4 different decisions. Nevertheless, there is an indication that the implicit-training procedure requires many more trials than does explicit training, since this was the case even when the implicit-training program had only half as many decisions to deal with as did the other programs.

Implicit training requires more trials because not only are training generalization techniques being utilized but also generalization techniques for determining variable relevancy. The important point, however, is that only a modest number of trials is required by either procedure to produce a program capable of playing a complex game, like draw poker, with roughly the same level of skill as an experienced human player.

NUMBER OF REDUNDANCIES. The third column of Table 5-3 contains the number of action rules made redundant during training. It is seen that more redundancies occurred during manual training than occurred during either automatic training or implicit training. One explanation is that the human trainer was less consistent during training than was the program trainer or axiom set and this inconsistancy led to an increase in the number of redundancies created. More important is the result that the modification and generalization techniques employed form learning systems which are quite stable and which accordingly create very few redundancies during the acquisition process.

NUMBER OF ACTION RULES. The fourth column of Table 5-3 contains the number of action rules either created by training or put into the system by hand. Note that although the trainee (the program containing the automatic-training heuristics) contained 9 fewer action rules than did its trainer (the program containing the built-in heuristics) it played almost as well as the trainer. Here the training process acted like a transformation procedure, changing a lengthy, thorough set of action rules into a compact, efficient set, leaving out rules corresponding to game situations seldom encountered in actual play.

The number of action rules created by the implicit-training process is seen to be less than the number created by explicit training. This difference is due simply to the fact that during implicit training the program had only four decisions to associate with game situations, while during the explicit training it had eight decisions. More generally speaking, it is seen from column 4 that a surprisingly small number of action rules (and associated bf rules) are needed to describe a thorough and effective set of heuristics for the game of draw poker.

PROGRAM PROFICIENCY. The fifth column of Table 5-3 contains the percent difference between the program's winnings and the opponent's winnings during an application of the proficiency test, expressed as a percentage of the amount won by the winning player. A plus percentage indicates that the program was the winning player, a minus percentage that the opponent was the winner. It is clear by comparing the difference in winnings before and after training that both the explicit and the implicit training procedures led to a significant increase in the playing ability of the programs involved.

However, the increase in playing ability during implicit training was not as great as the increase during explicit training. This result is due, presumably, to the following factors: (1) the axiom set, which provides a means for deducing "good" decisions, does not provide the program with decisions which are as shrewd or perceptive as those provided by a human trainer, (2) the program must use a complex generalization process to determine variable relevancy during implicit training, while it is given this information by the trainer during explicit training, and (3) the program is permitted to learn to make only half as many different decisions during implicit training as it can learn to make during explicit training.

CONVERGENCE. The last two columns of Table 5-3 contain a measure of the agreement obtained between (a) the trainer and trainee and (b) the axiom set and the implicit-training program, both before and after training. In each case the percentage is based on the number of identical decisions made during 50 consecutive game situations. It is seen from Table 5-3 that a high percentage of agreement or degree of convergence was achieved for both case (a) and case (b) above.

However, the degree of convergence for case (b) is less than that for case (a), probably because of the following aspect of the implicit-training procedure. The axiom set is used, together with the value of the opponent's hand, to logically deduce the decision that would have maximized the program's score, and this is considered by the program to be the decision it should have made during actual play. But during actual play the decisions of the program are based on a set of action rules which do not include the value of the opponent's hand (this value is unknown at the time).

For example, the "trainer" (the program as it performs deductions with the axiom set) may indicate that in game situation S action A should be taken and that in game situation S' action A' should be taken. If the only difference between S and S' is the value of the opponent's hand then the two situations are identical when put into action rule form. Thus it appears to the "trainee" (the program as it uses the action rules to make a decision) that the "trainer" is sometimes inconsistent, and as a result the percentage of agreement between the two is reduced.

CHAPTER 6

CONCLUSIONS

6.1 ACHIEVEMENTS

In the preceeding chapters a number of ideas relative to the problem of implementing machine learning of heuristics were presented and investigated. The achievements resulting from this examination of the problem will now be briefly summarized.

First, a method of representing heuristics (as production rules) was developed which facilitates dynamic manipulation of the heuristics by the program embodying them. This representation technique permits separation of the heuristics from the program proper, provides clear identification of individual heuristics and indicates how they are interrelated, makes the modification or replacement of heuristics a trivial task, and makes it simple to use the heuristics to obtain a decision from the system. Furthermore, a language for specifying heuristics was formulated which serves as a convenient intermediate step in the process of translating informally stated heuristics into production rules.

Second, procedures were developed which permit a problem-solving program employing heuristics in production rule form to learn to improve its performance by evaluating and modifying existing heuristics and hypothesizing reasonable new ones, either during a special training process or during normal program operation. These learning procedures are applicable in all cases where each of the

subvector variables, the program variables which directly influence or are influenced by the program's decisions, can be considered to have a range consisting of a set of integer values.

Third, the abovementioned representation and learning techniques were reformulated in the light of existing stimulus-response theories of learning, and five different S-R models of human heuristic learning in problem-solving environments were constructed and examined in detail. Experimental designs for testing these information processing models were also proposed and discussed.

Finally, the feasibility of using the aforementioned representation and learning techniques in a complex problem-solving situation was demonstrated by applying these techniques to the problem of making the bet decision in draw poker. This application, involving the construction of a computer program, demonstrated that (a) a surprisingly small number of production rules are needed to describe a set of heuristics for draw poker which enables a computer program to play the game with roughly the same level of skill as an experienced human player, (b) the program, whether learning via the training process or learning during normal program operation, requires only a modest number of acquisition trials to produce a thorough and effective set of heuristics for draw poker, and (c) the modification and generalization techniques which form the basis of the learning process lead to the creation of learning systems which are highly non-redundant or stable and whose decisions tend to converge to those supplied by the trainer during training.

6.2 AREAS FOR FUTURE INVESTIGATION

The ideas presented in the previous chapters suggest a number of areas which merit further investigation. These areas will now be specified and briefly discussed.

Learning the Decision Matrix

The learning system described in Chapters 3 and 5 which learns through actual game experience rather than explicit training must be supplied with a decision matrix. This matrix, it will be recalled, has a row corresponding to each decision the system can make and a column corresponding to each subvector variable. Each entry E_{ij} in the matrix indicates why the variable j is relevant, if when decision i is made the variable is in fact relevant. The next logical step in the process of expanding the power of the learning system is to eliminate the requirement that the system be supplied with a decision matrix. This can be accomplished by initially providing the system with an empty decision matrix and then having it learn through game experience what the entries in the matrix should be.

CHANGING LOGICAL OPERATORS. One approach to the problem of learning the decision matrix entries will now be outlined. As mentioned in Chapter 3 there are essentially two ways an action rule can be generalized upon to catch the symbolic subvector (or program subvector).

(1) Training Method: the sets corresponding to the symbolic values in the left part of the rule are enlarged by changing the numerical values in the predicates defining the sets.

(2) Hypothesis-formation Method: some of the relevant subvector variables (variables which have symbolic values other than the value *) in the left part of the rule are made irrelevant (are given the value *).

In order to implement the learning of the decision matrix entries a third method of modifying an action rule to catch the symbolic subvector is needed. This method is shown below.

(3) Decision-matrix Method: the logical operators in the predicates defining the sets corresponding to the symbolic values in the left part of the rule are changed, and each time a logical operator is changed the corresponding entry in the decision matrix is also changed in the same manner.

EXAMPLE. A simple illustration will serve to clarify this procedure. Assume the subvector is (P, B), the action rule to be modified is $(P1, B1) \rightarrow d_1$ where $P1 \rightarrow P$, $P > 15$ and $B1 \rightarrow B$, $B < 4$, the program subvector is (7, 2), and the current decision matrix is as shown below.

	P	B
d_1	>	<
d_2	<	>

Figure 6-1.

Then the action rule can be modified to catch the program subvector by changing the logical operator in the definition of P1 from > to < . Thus the definition becomes P1 → P, P < 17 . The numerical value in the definition is adjusted so that 16 is still a member of the set defined by P1. The entry in the decision matrix which corresponds to the logical operator just changed is the one found by entering the matrix at row d_1 , column P . Consequently, the decision matrix entry at this location is also changed and the matrix takes the form shown below.

	P	B
d_1	<	<
d_2	<	>

Figure 6-2.

If the decision matrix used by the learning system is initially empty the system can be thought of as hypothesizing whether > or < should be an entry at each location in the matrix and then later testing and revising each hypothesis.

CREDIT ASSIGNMENT. The crucial problem involved in using this approach to implement the decision matrix learning is the following. If method (1) is not applicable for modifying the action rule to catch the symbolic subvector, either method (2) or method (3) can be applied. The problem is to devise a priority scheme that specifies which of these two methods to use in any particular learning situation.

In a general sense, the problem is that of determining which of two concurrent hypotheses is to blame when an error is detected, the relevancy hypothesis or the decision-matrix hypothesis. This is another example of the credit-assignment problem, an extremely difficult and heretofore unsolved problem in artificial intelligence.

In this case, however, the priority scheme does not have to solve the problem single-handedly by determining with perfect accuracy which hypotheses are in error. It operates in conjunction with a learning system which is self-correcting, that is, which modifies or removes poor action rules. Thus the priority scheme need only be accurate enough to keep from overloading the self-correction mechanism, thereby permitting the learning system to converge at a reasonable speed.

Learning the Function Definitions

Another way to expand the power of the learning system is to require that it learn the function definitions. (They are ordinarily supplied to the system.) The functions (ff rules), described in Chapter 3, are defined by mathematical expressions composed of bookkeeping variables and function variables. Mathematical expressions of this type are a very compact, efficient way to represent heuristics, and for this very reason are quite difficult to manipulate or learn.

EXPANDING THE SUBVECTOR. Rather than trying to devise a system which will learn the function definitions directly, the following approach can be taken. Expand the subvector (the set of dynamic variables) by including in it all the bookkeeping variables needed to define the

functions. Then during the learning process described in Chapter 3 a number of action rules (and associated bf rules) will be learned which are roughly equivalent to the original action rules containing function definitions.

EXAMPLE. To see how a set of action rules can approximate a single action rule and its associated function definitions consider the following example. Assume that the subvector is (P, B) and the function A is defined as $A \rightarrow E + 3$, where E is a bookkeeping variable with a range of 1 to 6. Then the action rule and function definition

$$(P1, B1) \rightarrow (*, A)$$
$$A \rightarrow E + 3 \qquad \text{set 1}$$

can be approximated by the set of production rules given below, in which E is considered a subvector variable.

$$(P1, B1, E1) \rightarrow (*, 8, *)$$
$$(P1, B1, E2) \rightarrow (*, 6, *) \qquad \text{set 2}$$
$$(P1, B1, *) \rightarrow (*, 4, *)$$
$$E1 \rightarrow E,\ E > 4$$
$$E2 \rightarrow E,\ E > 2$$

The action advocated by set 2 is compared below to the action advocated by set 1.

	New value of B	
E	Set 1	Set 2
1	4	4
2	5	4
3	6	6
4	7	6
5	8	8
6	9	8

It is clear that set 2 does approximate set 1. In general, the number of action rules needed to approximate a function definition depends on the complexity of the function and the range of the function variables.

Other Areas of Interest

There are a number of areas remaining which, if properly exploited, could lead to an increase in the power of the proposed learning system. Two of these areas will now be briefly described.

IMPROVING THE AXIOM SET. One area which presents a challenge is the axiom set and associated deduction techniques used to supply the system with good decisions. In Chapter 5 it was noted that the degree of convergence exhibited by the learning system is reduced when the axiom set is used in place of a trainer. The explanation given for this was, in brief, that the axiom set has a tendency to appear inconsistent to the learning system, since in its deduction process

it makes use of the value of the opponent's hand, a variable which the learning system does not have available.

Since the value of the opponent's hand is essential to the axiom system operation and cannot be given to the learning system (at the time it makes a decision) an indirect approach to the problem is in order. A profitable approach might be to use a more sophisticated axiom set, one which has not only the goal of maximizing the program's score but also the goal of providing a decision which is reasonable when the value of the opponent's hand is unknown. However, this approach, in one sense, is more a restatement of the problem than a bona fide solution. As the axiom set is made more sophisticated the problem of finding a necessary and sufficient set of axioms becomes increasingly difficult.

DEFINING THE TASK ENVIRONMENT. Another area which presents a challenge is the problem of devising an effective way of defining the task environment in which the learning system operates. The task environment can be considered to consist of the set S of all possible situations which can occur and the set D of all possible decisions which can be made. This environment is defined by (1) specifying the subvector variables and their ranges, and (2) defining and partitioning the decision set. For example, the set D used in Chapter 5 is shown below.

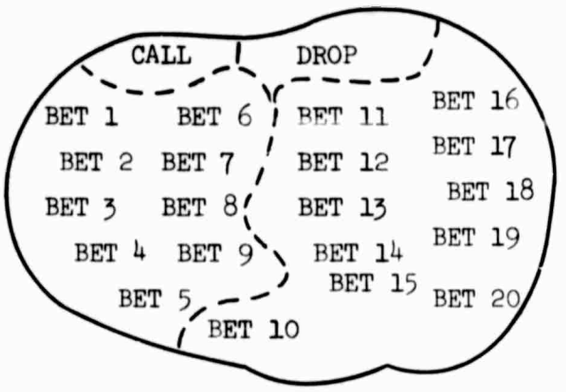


Figure 6-3.

The dotted lines in Figure 6-3 indicate how the set was partitioned into subsets.

During the learning process an ordered list of action rules is acquired which effectively partitions set S into n subsets, establishing a one-one correspondence between the subsets of S and and the subsets of D . It should be clear that the manner in which the subvector variables are chosen and defined (thus defining S) and the way in which the decision set is partitioned both have a profound influence on the prospective capabilities of the learning system.

To illustrate, consider the task of partitioning the decision set D . This set should ideally be partitioned to (a) maximize the speed of convergence of the learning system, and (b) permit the system to become proficient at the problem-solving task being learned. An approach to maximizing convergence speed is to generate trial partitionings. Each partitioning restructures or redefines the trainer's decision space, and each newly-defined decision space can be used to

estimate the resulting speed of convergence of the system. The size of this estimate can be used as one of the criteria for determining a good partitioning of D. Another criterion can be the number of subsets D is partitioned into, where the assumption is that potential proficiency increases with the number of subsets used.

The speed of convergence can be estimated by sampling the decision space of the trainer to determine the approximate number ans size of decision clusters in the space. Since (1) the number of action rules needed to describe the space is roughly equal to the number of clusters in the space, and (2) the optimal generalization constant K is very nearly equal to the average cluster width, this sampling provides an estimate of the speed of convergence of the learning system.

BLANK PAGE

BIBLIOGRAPHY

Anonymous, 1967. Heuristic programs and algorithms, SICART Newlsetter, November, pp. 10-25.

Baumann, R., Feliciano, M., Bauer, F., and Samelson, K. 1964. Introduction to ALGOL, Prentice Hall, Inc., Englewood Cliffs, N. J.

Bernstein, A., and Roberts, M. 1958. Computer vs. chess player. Scientific American, Jund, v. 198, pp. 96-105.

Black, F. 1964. A deductive question answering system. Doctoral dissertation, Harvard University, Cambridge, Mass.

Bower, G. H. 1966. Mathematical learning theory. Theories of Learning, Hilgard, E. R., and Bower, G. H., Chapter 11, Appleton-Century-Crofts, New York.

Bruner, J. W., Goodnow, J. J., and Austin, G. A. 1956. A Study of Thinking, Wiley, New York.

Bush, R. R., and Mosteller, F. 1955. Stochastic Models for Learning, Wiley, New York.

Carne, E. B. 1965. Artificial Intelligence Techniques, Spartan Books, Inc., Washington, D. C.

Chomsky, N. 1959. On certain formal properties of grammars. Information and Control, v. 2 pp. 137-167.

Davis, M., and Putnam, H. 1960. A computing procedure for quantification theory. ACM Journal v. 7, no. 2, pp. 201-215.

Ekman, T., and Froberg, C. 1965. Introduction to ALGOL Programming, Studentlitteratur, Lund, Sweden.

Estes, W. K. 1959. The statistical approach to learning theory. Psychology: A study of a Science, Vol. 2 S. Koch (ed.), McGraw-Hill, New York.

Feigenbaum, E. A. 1959. An information processing theory of verbal learning. Rand Corporation Paper P-1817, October, Santa Monica, Calif.

Feigenbaum, E. A. 1963. The simulation of verbal learning behavior. Computers and Thought, Feigenbaum, E. A. and Feldman, J. (eds.) pp. 297-309.

Feigenbaum, E. A. and Feldman, J. 1963. Computers and Thought, New Yord, McGraw-Hill.

Feigenbaum, E. A., and Simon, H. A. 1964. An information processing theory of some effects of similarity, familiarization, and meaningfulness in verbal learning. Journal of Verbal Learning and Verbal Behavior, v. 3, no. 5, October, pp. 385-396.

Feigenbaum, E. A. 1967. Information processing and memory. Fifth Berkeley Symposium on Mathematical Statistics and Probability, v. 4, pp. 37-51, University of California, Berkeley, Calif.

Feldman, J. 1963. Simulation of behavior in the binary choice experiment. Computers and Thought, Feigenbaum, E. A., Feldman, J. (eds.) McGraw-Hill, New York, pp. 329-346.

Feldman, J., Tonge, F., and Kanter, H. 1963. Empirical explorations of a hypothesis-testing model of binary choice behavior. Symposium on Simulation Models: Methodology and Applications to the Behavioral Sciences, Hoggatt, A., and Balderston, F. (eds.) Cincinnati, Ohio, South-Western Publishing Co., pp. 55-100.

Friedberg, R. M. 1958. A learning machine, part I. IBM Journal, June v. 3, pp. 282-287.

Friedberg, R. M., Dunham, B., and North, J. H. 1959. A learning machine, part II. IBM Journal, June, v. 3, pp. 282-287.

Gelernter, H. 1959. Realization of a geometry theroem-proving machine. Proceedings of the International Conference on Information Processing, Paris, UNESCO House, pp. 273-282.

Gelernter, H., Hansen, J. R., and Loveland, D. W. 1960. Empirical explorations of the geometry theorem-proving machine. Proceedings of the Western Joint Computer Conference, May 1960, pp. 143-147.

Green, B. F. 1963. Digital Computers in Research, McGraw-Hill, New York.

Hilgard, E. R., and Bower, G. H. 1966. Theories of Learning, Appleton-Century-Crofts, New York.

Hunt, E. B. 1962. Concept Learning: An Information Processing Problem, John Wiley & Sons, New York.

Hunt, E. B., Marin, J., and Stone, P. J. 1966. Experiments in Induction, Academic Press, New York.

Ingerman, P. Z. 1966. A Syntax-Oriented Translator, Academic Press, New York.

Irons, E. T. 1961. A syntax directed compiler for ALGOL 60. ACM Communications, v. 4, January, pp. 51-55.

Irons, E. T. 1963. The structure and use of the syntax directed compiler. Annual Review in Automatic Programming, v. 3, Goodman, R. (ed.) MacMillan Co., New York, pp. 207-228.

Irons, E. T. 1964. Structural connections in formal languages. ACM Communications, v. 7, no. 2, February, pp. 67-71.

Kister, J., Stein, P., Ulam, S., Walden, W., and Wells, M. 1957. Experiments in chess, ACM Journal, April v. 4, no. 2, pp. 174-177.

Kochen, M. 1960. Experimental study of hypothesis formation by computer. IBM Report RC-294, International Business Machines Corporation, Yorktown Heights, New York.

Kochen, M. 1961. An experimental program for the selection of disjunctive hypotheses. Proceedings of the Western Joint Computer Conference, v. 19, pp. 571-578.

McCarthy, J. 1959. Programs with common sense. Proceedings of the Symposium on Mechanisation of Thought Processes, National Physical Laboratory, Teddington, England, Blake, D., and Uttley, A. (eds.) pp. 75-84.

McCarthy, J. 1962. Towards a mathematical science of computation. Proceedings ICIP.

McCarthy, J. 1962. LISP 1.5 Programmer's Manual, MIT Press, Cambridge, Massachusetts.

McCarthy, J. 1965. Problems in the theory of computation. Proceedings of the IFIP Congress, pp. 219-222.

Minsky, M. L. 1961. Steps toward artificial intelligence. Proceedings of the IRE, v. 49, no. 1, January.

Newell, A., and Simon, H. A. 1956. The logic theory machine. IRE Transactions on Information Theory, v. IT-2, no. 3, pp. 61-79.

Newell, A., Shaw, J. C., and Simon, H. A. 1957a. Empirical explorations of the logic theory machine. Proceedings of the Western Joint Computer Conference (WJCC) pp. 218-239.

Newell, A., and Shaw, J. C. 1957b. Programming the logical theory machine. Proceedings of the Western Joint Computer Conference (WJCC), pp. 230-240.

Newell, A., Shaw, J. C., and Simon, H. A. 1958. Chess-playing programs and the problem of complexity. IBM Journal of Research and Development, v. 2, no. 4, pp. 320-335.

Newell, A., Shaw, J. C., and Simon, H. A. 1959. Report on a general problem-solving program. Rand Corporation Paper P-1584, Santa Monica, California.

Newell, A., and Simon, H. A. 1961. GPS, a program that simulates human thought. Rand Corporation Paper P-2257, Santa Monica, California.

Newell, A. 1962. Some problems of basic organization in problem-solving programs. Rand Report RM-3283-PR, December, Santa Monica, California, p. 8.

Newell, A., and Ernst, G. 1965. The search for generality. Proceedings of the IFIPS Congress 65, v. 1, pp. 17-24.

Newell, A. 1966. On the analysis of human problem solving protocols. Proceedings International Symposium on Mathematical and Computational Methods in the Social Sciences.

Newell, A. 1967. Studies in problem solving: subject 3 on the crypt-arithmetic task Donald + Gerald = Robert. Center for the Study of Information Processing, Carnegie Institute of Technology, Pittsburgh, Pa.

Rapoport, A. 1966. Two-Person Game Theory, University of Michigan Press, Ann Arbor, Michigan.

Reitman, W. R. 1965. Cognition and Thought, An Information Processing Approach, Wiley, New York.

Robinson, G. A., Wos, L. T., and Carson, D. F. 1964. Some theorem-proving strategies and their implementation. AMD Technical Memo no. 72, Argonne National Laboratory.

Robinson, J. A. 1965a. A machine-oriented logic based on the resolution principle. ACM Journal, v. 12, January, pp. 23-41.

Robinson, J. A. 1965b. Automatic deduction with hyper-resolution. International Journal of Computer Mathematics, v. 1, July , pp. 227-234.

Rosenblatt, F. 1958. The perceptron: a probabilistic model for information storage and organization in the brain. Psychol. Rev., v. 65, pp. 386-408.

Rosenblatt, F. 1962. Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms, Spartan Press, New York.

Samuel, A. L. 1959. Some studies in machine learning using the game of checkers. IBM Journal, v. 3, no. 3, pp. 210-229.

Samuel, A. L. 1960. Programming computers to play games. Advances in Computers Vol. 1, Alt. F. L. (ed.), Academic Press, New York, pp. 165-192.

Samuel, A. L. 1967. Some studies in machine learning using the game of checkers, II - Recent Progress. IBM Journal, November, v. 11, no. 6, pp. 601-617.

Selfridge, O. G. 1959. Pandemonium: a paradigm for learning. Proc. Symposium on Mechanisation of Thought Processes, H. M. Stationery Office, London.

Shannon, C. E. 1950. Programming a digital computer for playing chess. Philosophy Magazine, March, v. 41, pp. 356-375.

Simon, H. A. 1961. Experiments with a heuristic compiler. Rand Paper, P-2349, Santa Monica, California.

Simon, H. A. 1963. The heuristic compiler. Rand Corporation Report, RM-3588-PR, Santa Monica, California.

Simon, H. A. and Kotovsky, K. 1963. Human acquisition of concepts for sequential patterns, Psychol. Rev. vol. 70, no. 6, pp. 534-546.

Slagle, J. R. 1961. A computer program for solving problems in freshman calculus. Doctorial Dissertation, MIT, Cambridge, Massachusetts.

Slagle, J. R. 1963. Game trees, m & n minimaxing, and the m & n alpha beta procedure. Lawrence Radiation Laboratory AI Report no. 3, November, Livermore, California.

Slagle, J. R. 1967. Automatic theorem proving with renamable and semantic resolution. ACM Journal, October, v. 14, no. 4, pp. 687-697.

Slagle, J. R., and Bursky, P. 1968. Experiments with a multipurpose, theorem-proving heuristic program. ACM Journal, v. 15, no. 1, pp. 85-99.

Stefferud, E. 1963. The logic theory machine: a model heuristic program. Rand Corporation report RM-3731-CC, Santa Monica, California.

Tonge, F. M. 1961. A Heuristic Program for Assembly Line Balancing, Englewood Cliffs, N. J., Prentice-Hall.

Trakhtenbrot, B. A. 1963. Algorithms and Automatic Computing Machines, D. C. Heath and Company, Boston.

Turing, A. M. 1950. Computing machinery and intelligence. Mind, October, v. 59, pp. 433-460.

Uhr, L., and Vossler, C. 1961. A pattern recognition program that generates, evaluates, and adjusts its own operators. Teleological mechanisms, Annals of the New York Academy of Science, v. 50, no. 189, pp. 555-569.

Wang, H. 1960a. Toward mechanical mathematics. IBM Journal of Research and Development, v. 4, no. 1., pp. 2-22.

Wang, H. 1960b. Proving theorems by pattern recognition-I. ACM Communications, v. 3, April 1960, pp. 220-234.

Wang, H. 1961. Proving theorems by pattern recognition-II. Bell System Technical Journal, v. 40, Jan. 1961, pp. 1-42.

Wirth, N., and Weber, H. 1966. EULER: a generalization of ALGOL, and its formal definition: part I. ACM Communications, v. 9 no. 1, January, pp. 13-25.

Wirth, N., and Weber, H. 1966. EULER: a generalization of ALGOL, and its formal definition: part II. ACM Communications, v. 9, no. 2, February, pp. 89-99.

Wos, L., Carson, D., and Robinson, G. 1964. The unit preference strategy in theorem proving. AFIPS Conference Proceedings, v. 26, Spartan Books, Washington, D. C., pp. 615-621.

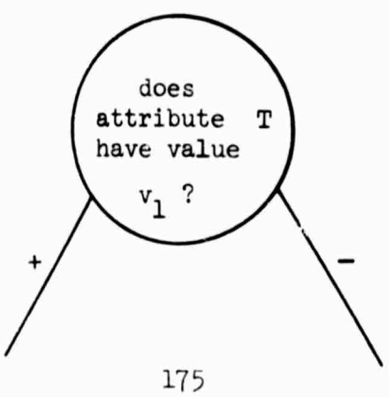
APPENDIX A

MODELS OF STRATEGY LEARNING

I. Generalization Technique for Growing Concept Trees

The tree-growing technique discussed in section 4.2 is summarized below. This technique is applied to the current unordered list of S-A connections.

1. Group the situation descriptions (or S's) into sets determined by the actions associated with them, i.e., all the S's connected to action Ai form a set called Ai . The situation descriptions comprising all these sets will be called the class of relevant S's .
2. If all the S's in the class of relevant S's are members of one set then grow a terminal node containing the name of that set.
3. If it is not the case that all the S's in the class of relevant S's are members of one set then grow a test node using as the test the attribute value determined by the procedure described below. Eliminate from consideration any value which occurs in every S of every set. This test node has the form:



4. If a test node was grown in step 3, sort all the S's in the current class of relevant S's down the node to either the positive side or the negative side. However, if an S has * as the value of the test attribute T than sort it down both sides of the node. Now take all S's which sorted down the positive branch and apply steps 1 through 4 again, using these S's as the current class of relevant S's and growing the next node from this positive branch. Finally, take all S's which sorted down the negative branch and apply steps 1 through 4 again, using these S's as the current class of relevant S's and growing the next node from this negative branch.

CHOOSING ATTRIBUTE VALUES. The attribute value to use as a test at a node (see step 3 above) is ascertained by applying the following procedure to the sets which partition the current class of relevant S's :

(a) For each attribute value calculate the maximum value of bc , the value of av , and the value of sv . For a particular set containing attribute value v_1 of attribute T ,

$$bc = \frac{\text{(number of times } v_1 \text{ occurs as a value of T in the set)}}{\text{(total number of S's in the set)}}$$

The maximum value of bc for attribute value v_1 is just the largest value obtained when the above formula is applied to every set. The quantities av and sv are defined as follows for attribute value v_1 of attribute T .

$$av = \frac{\begin{array}{l}(\text{the number of sets where} \\ \text{* is used at least once} \\ \text{as the value of T})\end{array} + \begin{array}{l}(\text{the total number of *'s} \\ \text{used as the value of T,} \\ \text{counting all sets})\end{array}}{(\text{total number of S's in all the sets}}$$

$$sv = \frac{\begin{array}{l}(\text{number of times } v_1 \text{ occurs as a value of T in all sets} \\ \text{except the set used to determine the maximum value of bc})\end{array}}{(\text{total number of S's in all the sets}}$$

(b) Choose as the test at the node that attribute value which maximizes the arithmetic expression ae , where ae = bc-av-sv . If more than one value maximizes ae , one of them could be selected at random. Instead, however, select one according to some arbitrary deterministic criterion, such as h's before, p's , p's before b's , and in case of a tie on letters, low digits before high digits.

This procedure leads to the selection of tests which tend to minimize the size of the tree being grown. This is because the procedure favors tests on values which occur often in one set but seldom in all other sets, a condition conducive to minimal tree generation.

EXAMPLE OF TREE GROWING. To clarify this tree-growing procedure to above rules will be applied to the list of S-A connections shown below. The attributes considered are H , P , and B .

h1,*,b2 → A1

h1,p2,* → A1

h2,p2,b2 → A2

h1,p1,b1 → A3

h2,p1,b2 → A3

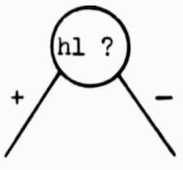
h2,*,b1 → A4

h3,*,* → A4

The S's are grouped into sets as indicated (step 1):

A1	A2	A3	A4
h1,*,b2	h2,p2,b2	h1,p1,b1	h2,*,b1
h1,p2,*		h2,p1,b2	h3,*,*

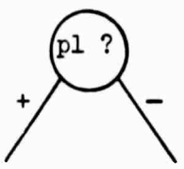
Since the S's are not all members of one set, a terminal node is not grown (step 2). Instead, a test node is grown (step 3) using h1 as the test, since for h1 the maximum bc is $\frac{2}{2}$ or 1 (from set A1), av is $\frac{0+0}{7}$ or 0, and sv is $\frac{1}{7}$, and these values for bc, av, and sv produce the largest ae. The value of ae for h1 is thus $1-0-\frac{1}{7}$ or $\frac{6}{7}$, while the value of ae for the other attribute values is less. All the S's are sorted down the test node (step 4) to produce the following result:



A1	A3	A2	A3	A4
h1,*,b2	h1,p1,b1	h2,p2,b2	h2,p1,b2	h2,*,b1
				h3,*,*

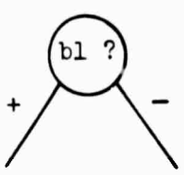
Now steps 1 through 4 are applied to the S's that sorted down the positive branch of the h1 test. This leads to the growing of a new test node. Since the value of ae is $1-\frac{2}{3}-0$ or $\frac{1}{3}$ for both p1 and b1

and is $\frac{1}{2}\frac{2}{3}-0$ or $\frac{1}{6}$ for both p2 and b2 the test is made on either p1 or b1 (in this case p1, since a priority of p's before b's has been established). The attribute value h1 is not considered since it appears in every S of every set being currently processed. Since p1 is picked as the test at this node, after the S's are sorted down the node the result is:



A1	A3	A1
h1,*,b2	h1,p1,b1	h1,*,b2
		h1,p2,*

Now steps 1 through 4 are applied to the S's that sorted down the positive branch of the p1 test, and a test node based on either b1 or b2 must be grown. The attribute values h1 and p1 are not considered since they appear in every S of every set being currently processed. Value b1 is picked as the test (since a priority of low digits before high digits has been established) and the S's are sorted down the node, resulting in:



A3	A1
h1,p1,b1	h1,*,b2

Now steps 1 through 4 are applied to the S's that sorted down the positive branch of the b1 test, but since all the S's belong to one set, a terminal node is grown (step 2) containing A3 . Similarly, when steps 1 through 4 are applied to the negative branch of the b1 test a terminal node containing A1 is grown. Then these steps are applied to the negative branch of the p1 test and another terminal node containing A1 is grown. Finally steps 1 through 4 are applied to the negative branch of the h1 test, and three more test nodes plus four terminal nodes are grown. The complete tree is shown below.

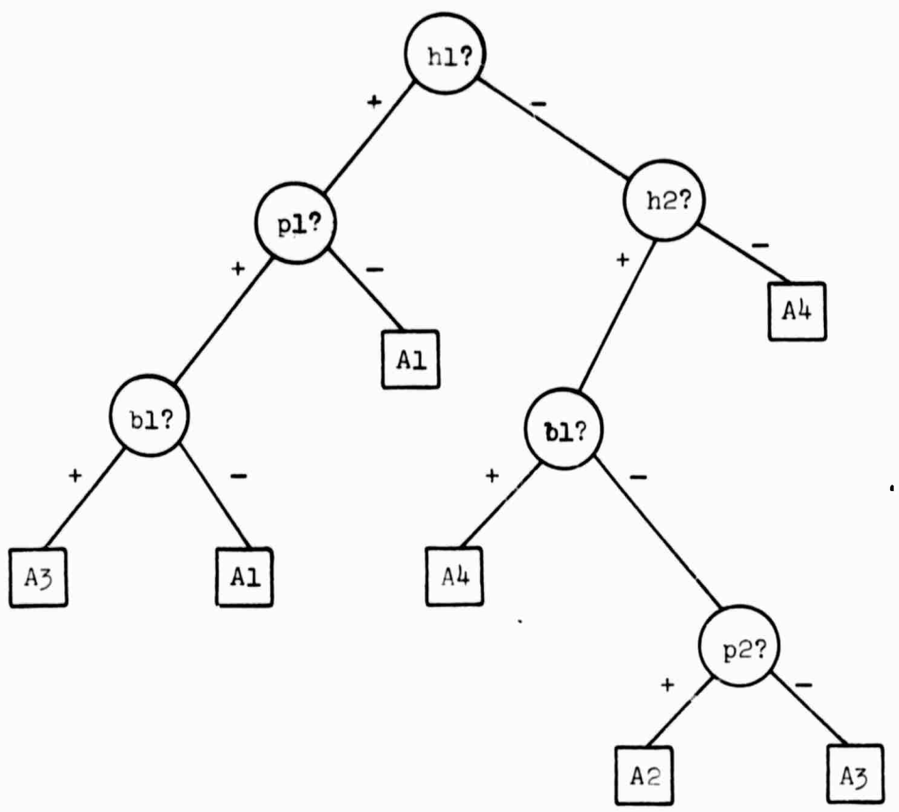


Figure A-1.

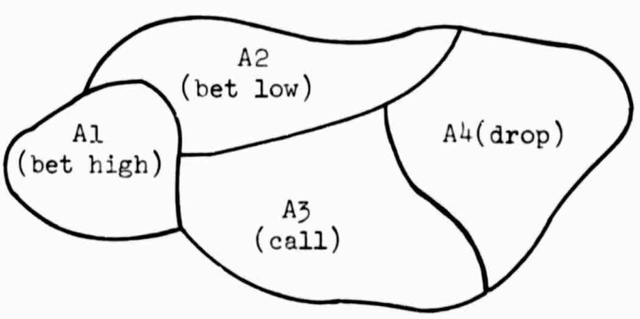
It is easily demonstrated that all the S's from the original S-A connection list sort down the tree into terminal nodes corresponding to the actions with which they were associated.

II. A Game-Playing Interpretation of the Environment Defined in Figure 4-3.

The game under consideration here is an extremely simplified version of draw poker where H is the value of your hand, P the amount of money in the pot, and B the amount last let by the opponent.

Attributes:	H(hand)	P(pot)	P(opponent's last bet)
Range of Values:	1 - 50	1 - 60	1 - 10
Abstract Values:	h1(good) h2(fair) h3(poor)	p1(large) p2(small)	b1(large) b2(small)

Universe of Situations:



Heuristics:
hand-good and bet-small → bet high
hand-good and pot-small → bet high
hand-fair and pot-small and bet-small → bet low
hand-good and pot-large and bet-large → call
hand-fair and pot-large and bet-small → call
hand-fair and bet-large → drop
hand-poor → drop

APPENDIX B

HEURISTICS FOR DRAW POKER

I. Definition of the Game

In the version of draw poker being considered a game consists of a predetermined number of rounds-of-play between two players. Each round-of-play (r-o-p) is comprised of the following sequence of events.

(1) Deal: Each player receives 5 cards (a hand) and antes 1 chip into the pot. The cards are dealt "face down", that is, each player sees only his own hand.

(2) Betting Interval: Each player alternately has the option of betting, calling, or dropping. A call terminates the betting interval and a drop terminates the round-of-play.

(3) Replace: Each player may remove from 0 to 3 cards from his hand and receive new cards to replace them.

(4) Betting Interval: Each player alternately has the option of betting, calling, or dropping. As before, a call terminates the betting interval and a drop terminates the round-of-play.

(5) Showdown: Both players display their hands, and the one with the highest ranking hand wins the money in the pot.

Betting is defined as placing in the pot an amount of money larger than the amount last placed there by the opposing player. The term "bet" stands for the difference between the amount placed in the pot and the amount previously placed there by the opponent. (In the standard poker jargon this is usually called the raise rather than the bet.) Only integer bets of from 1 to 20 are allowed.

A call is defined as placing in the pot an amount of money equal to the amount last placed there by the opposing player. Thus a call can be thought of as a bet of zero. A call always terminates the

betting interval and after cards have been replaced leads directly to the showdown. However, a call may not be made until a bet has been made in the current betting interval.

A drop is defined as withdrawing from the present round-of-play relinquishing all money in the pot to the opposing player. No hands are displayed when a player drops. All the standard poker hands from one-of-a-kind to a royal flush are recognized, but no wild cards are permitted.

II. Informal Description of the Bet Decision Heuristics

The heuristics used by the computer program in making the bet decision in draw poker are listed below.

1. A player with a hand that is sure to win should bet the largest amount possible without causing the opponent to drop. However, if the pot is extremely large a call should be made.

2. A player with a hand that has an excellent chance of winning should bet the largest amount possible without causing the opponent to drop. However, a call should be made after the pot becomes quite large.

3. A player with a hand that has a good chance of winning should bet a medium amount, unless the opponent is easily bluffed and cards have not yet been replaced. In this case a small bet should be made. However, if either the pot becomes quite large or both the pot and the opponent's last bet are fairly large then a call should be made. The call should be made sooner if the opponent replaces fewer than 2 cards or has not yet replaced cards. Furthermore, a call should be made if the opponent is a conservative player and replaces two cards.

4. A player with a hand that has a poor chance of winning should call, unless the opponent has not yet bet. In this situation a small bet should be made. However, if cards have been replaced, the opponent's last bet is large, and the pot-bet ratio is small a drop should be made. Furthermore, if the pot and the opponent's last bet are small, and the opponent is easily bluffed a bluff bet (a large bet) should be made. But if the opponent is a conservative player and replaces 0 or 2 cards and the pot-bet ratio is large, a call should be made.

5. A player with a hand that has almost no chance of winning should drop unless both the pot and the opponent's last bet are very small. In this case a small bet should be made if the opponent has not yet bet or a call made if the opponent has bet and the pot-bet ratio is large. However, if the opponent is very easily bluffed and replaces 3 cards, and both the pot and the opponent's last bet are small then a bluff bet (a fairly large or a very large bet) should be made.

6. A hand is sure to win if its value is large, and is very much larger than the expected value of the opponent's hand.

7. A hand has an excellent chance of winning if its value is not large, but is very much larger than the expected value of the opponent's hand.

8. A hand has a good chance of winning if its value is much larger than the expected value of the opponent's hand.

9. A hand has a poor chance of winning if its value is only slightly larger than the expected value of the opponent's hand.

10. A hand has almost no chance of winning if its value is not larger than the expected value of the opponent's hand.

11. The expected value of the opponent's hand decreases as the average bet made during an r-o-p times 'the number of bets made by the opponent during an r-o-p' times 'the number of times the opponent was caught bluffing during the r-o-p' increases.

12. The probability that the opponent is bluffing increases as 'the number of times the opponent was caught bluffing' increases and decreases as 'a measure of conservative style by the opponent' increases.

13. A measure of conservative style by the opponent increases as 'a measure of the correlation between the opponent's hands and bets' and 'the number of times the opponent has dropped' increase.

14. The probability of being able to bluff the opponent increases as 'a measure of conservative style by the opponent' increases and decreases as 'the expected value of the opponent's hand' increases.

15. The largest bet possible without causing the opponent to drop increases as 'the probability of being able to bluff the opponent' decreases.

16. A small bet is one ranging from 1 to 5 .

17. A medium bet is one ranging from 3 to 9 .

18. A fairly large bet is one ranging from 10 to 15 .

19. A large bet is one ranging from 8 to 14 .

20. A very large bet is one ranging from 14 to 20 .

III. LASH Description of the Bet Decision Heuristics

The heuristics used by the computer program in making the bet decision in draw poker are presented below in LASH.

```
begin  'CALL' : POT ← POT+(2×LASTBET);LASTBET ← (0),
       'BETLAP' : POT ← POT+(2×LASTBET);LASTBET ← (LAP),
       'BETSB' : POT ← POT+(2×LASTBET);LASTBET ← (SB),
       'BETMB' : POT ← POT+(2×LASTBET);LASTBET ← (MB),
       'BETBB' : POT ← POT+(2×LASTBET);LASTBET ← (BB),
       'BETBBS' : POT ← POT+(2×LASTBET);LASTBET ← (BBS),
       'BETBBL' : POT ← POT+(2×LASTBET);LASTBET ← (BBL),
       'DROP' : VDHAND ← (0); LASTBET ← (0) .

if  H ≡ SW  then
   (if  P > 80 ∧ B≠0 then 'CALL' else 'BETLAP') otherwise
if  H ≡ EC  then
   (if  P > K1 ∧ B≠0 then 'CALL' else 'BETLAP') otherwise
if  H ≡ GC  then
   (if  P > K2 ∧ B≠0  ∧ (R=0∨R=1) then 'CALL  else
      (if  P > 15 ∨ B > 7 ∧ (R=0 ∨ R=1) then 'CALL' else
         (if  B≠0 ∧ R=2 ∧ OCS > K3 then 'CALL' else
            (if  P > K4 ∧ B≠0 ∧ R < 0 then 'CALL' else
               (if  BFO > K5 ∧ R < 0 then 'BETSB' else
                  (if  P > K6 ∧ B≠0 then 'CALL' else
                     (if  P < 15 ∧ B > 10 then 'CALL' else 'BETMB')))))))otherwise

if  H ≡ PC  then
   (if  B≠0 ∧ PB > 1 ∧ R=0 then 'CALL' else
      (if  B≠0 ∧ PB > 1 ∧ R=2 ∧ OCS > K7 then 'CALL' else
         (if  P < K14 ∧ B < 5 ∧ B≠0 ∧ BFO > K5 ∧ PB > 3 ∧ R≠-1 then 'BETBB' else
            (if  P < K9 ∧ B < K10 ∧ BFO > K11 then 'BETBB' else
               (if  B > 9 ∧ PB < 2 ∧ R≠-1 then 'DROP' else
                  (if  B≠0 then 'CALL' else 'BETSB'))))))) otherwise

if  H ≡ NC  then
   (if  R=0 then 'DROP' else
      (if  R=2 ∧ OCS > K12 then 'DROP' else
         (if  P < 13 ∧ B < 5 ∧ B≠0 ∧ BFO > K5 ∧ R=3 then 'BETBBS' else
            (if  P < K14 ∧ B < K15 ∧ BFO > K16 ∧ R≠-1 then 'BETBBL' else
               (if  B≠0 ∧ PB > K17 then 'CALL' else
                  (if  P < K32 ∧ B < 5 ∧ B≠0 then 'CALL' else
                     (if  P > K32 ∧ B < K13 then 'BETSB' else
                        (if  P < K14 ∧ B < K13 ∧ R≠-1 then 'BETSB' else 'DROP'))))))))).
```

```
SW is an  H  such that  (H-OH > K18 ∧ H ≥ K19),
EC is an  H  such that  (H-OH > K18 ∧ H ≤ K19),
GC is an  H  such that  (K20 < H-OH ∧ H-OH ≤ K18),
PC is an  H  such that  (K21 < H-OH ∧ H-OH ≤ K20),
NC is an  H  such that  (H-OH ≤ K21),
OH equals K22-(K23 x OAVGBET x OTBET x OB),
OB equals (K24 x OBLUFFS) - (K25 x CS),
CS equals (K26 x OCORREL) + (K27 x OD),
BO equals (K28 x CS) - (K29 x OH),
LAP equals K30 - (K31 x BO),
SB equals random (1,5),
MB equals random (3,9),
BBS equals random (10,15),
BB equals random (8,14),
BBL equals random (14,20),
H is a VDHAND such that (VDHAND > 0),
P is a POT such that (POT > -1),
B is a LASTBET such that (LASTBET) ≥ 0 ∧ LASTBET < 21),
BFO is a BLUFFO such that (BLUFFO ≥ 0 ∨ BLUFFO < 0),
PB is a POTBET such that (POTBET ≥ 0),
R is an ORP such that (ORP > -1 ∧ ORP < 4),
OCS is an OSTYLE such that (OSTYLE ≤ 0 ∨ OSTYLE < 0) end.
```

It is clear that a one-to-one correspondence exists between the first five informally stated heuristics in Appendix B, Part II (the heuristic rules) and the five major if-statements in the above routine. Similarly, there is one definition above for each of the other informally stated heuristics (the heuristic definitions). The last seven definitions given above (one for each subvector variable) do not correspond to any of the informal heuristics. Instead, they correspond somewhat to those game rules which define the allowable values for the game variables.

IV. Production Rule Description of the Bet Decision Heuristics

The production rules which correspond to the LASH routine shown in Appendix B, Part III are presented below. The first 62 rules are separated into five groups, each group having been generated from one of the five major LASH if-statements. The remaining rules correspond, in a one-to-one fashion, to the definitions set forth in the LASH routine.

1.	a.	(SW P8 B5 * * * *)	→	(* POT+(2xLASTBET) 0 * * * *)	call
	b.	(SW * * * * * *)	→	(* POT+(2xLASTBET) LAP * * * *)	bet
2.	a.	(EC P1 B5 * * * *)	→	(* POT+(2xLASTBET) 0 * * * *)	call
	b.	P1	→	P, P > K1	bf
	c.	B5	→	B, B > 0	bf
	d.	(EC * * * * * *)	→	(* POT+(2xLASTBET) LAP * * * *)	bet
3.	a.	(GC P2 B5 * * OR1 *)	→	(* POT+(2xLASTBET) 0 * * * *)	call
	b.	P2	→	P, P > K2	bf
	c.	OR1	→	R, R = 0 or 1	bf
	d.	(GC P9 B6 * * OR1 *)	→	(* POT+(2xLASTBET) 0 * * * *)	call
	e.	P9	→	P, P > 15	bf
	f.	B6	→	B, B > 7	bf
	g.	(GC * B5 * * OR2 CS1)	→	(* POT+(2xLASTBET) 0 * * * *)	call
	h.	OR2	→	R, R = 2	bf
	i.	CS1	→	OCS, OCS > K3	bf
	j.	(GC P3 B5 * * OR3 *)	→	(* POT+(2xLASTBET) 0 * * * *)	call
	k.	P3	→	P, P > K4	bf
	l.	OR3	→	R, R = -1	bf
	m.	(GC * * BO1 * OR3)	→	(* POT+(2xLASTBET) SB * * * *)	bet
	n.	BO1	→	BFO, BFO > K5	bf
	o.	(GC P4 B5 * * * *)	→	(* POT+(2xLASTBET) 0 * * * *)	call
	p.	P4	→	P, P > K6	bf
	q.	(GC P9 B7 * * * *)	→	(* POT+(2xLASTBET) 0 * * * *)	call
	r.	B7	→	B, B > 10	bf
	s.	(GC * * * * * *)	→	(* POT+(2xLASTBET) MB * * * *)	bet
4.	a.	(PC * B5 * PB2 OR4 *)	→	(* POT+(2xLASTBET) 0 * * * *)	call
	b.	PB2	→	PB, PB > 1	bf
	c.	OR4	→	R, R = 0	bf
	d.	(PC * B5 * PB2 OR2 CS2)	→	(* POT+(2xLASTBET) 0 * * * *)	call
	e.	CS2	→	OCS, OCS > K7	bf
	f.	(PC P6 B9 BO1 PB3 OR6 *)	→	(* POT+(2xLASTBET) BB * * * *)	bet
	g.	P6	→	P, P < K14	bf
	h.	B9	→	B, B < 5 ∧ B ≠ 0	bf
	i.	PB3	→	PB, PB > 3	bf
	j.	OR6	→	R, R ≠ -1	bf
	k.	(PC P5 B2 BO2 * * *)	→	(* POT+(2xLASTBET) BB * * * *)	bet
	l.	P5	→	P, P < K9	bf
	m.	B2	→	B, B < K10	bf
	n.	BO2	→	BFO, BFO > K11	bf
	o.	(PC * B8 * PB4 OR6 *)	→	(0 * 0 * * * *)	drop
	p.	B8	→	B, B > 9	bf
	q.	PB4	→	PB, PB < 2	bf
	r.	(PC * B5 * * * *)	→	(* POT+(2xLASTBET) 0 * * * *)	call
	s.	(PC * * * * * *)	→	(* POT+(2xLASTBET) SB * * * *)	bet
5.	a.	(NC * * * * OR4 *)	→	(0 * 0 * * * *)	drop
	b.	(NC * * * * OR2 CS3)	→	(0 * 0 * * * *)	drop
	c.	CS3	→	OCS, OCS > K12	bf
	d.	(NC P10 B9 BO1 * OR7 *)	→	(* POT+(2xLASTBET) BBS * * * *)	bet

e.		P10	→	P, P < 13	bf
f.		OR7	→	R, R = 3	bf
g.	(NC P6 B4 BO3 * OR6 *)		→	(* POT+(2xLASTBET) BBL * * * *)	bet
h.		P6	→	P, P < K14	bf
i.		B4	→	B, B < K15	bf
j.		BO3	→	BFO, BFO > K16	bf
k.	(NC * B5 PB1 * *)		→	(* POT+(2xLASTBET) 0 * * * *)	call
l.		PB1	→	PB, PB > K17	bf
m.	(NC P7 B9 * * * *)		→	(* POT+(2xLASTBET) 0 * * * *)	call
n.		P7	→	P, P < K32	bf
o.	(NC P7 B3 * * * *)		→	(* POT+(2xLASTBET) SB * * * *)	bet
p.		B3	→	B, B < K13	bf
q.	(NC P6 B3 * * OR6 *)		→	(* POT+(2xLASTBET) SB * * * *)	bet
r.	(NC * * * * * *)		→	(0 * 0 * * * *)	drop
6.		SW	→	H, H - OH > K18 and H ≥ K19	bf
7.		EC	→	H, H - OH > K18 and H < K19	bf
8.		GC	→	H, K20 < H - OH ≤ K18	bf
9.		PC	→	H, K21 < H - OH ≤ K20	bf
10.		NC	→	H, H - OH ≤ K21	bf
11.		OH	→	K22 - (K23 x OAVGBET x OTBET x OB)	ff
12.		OB	→	(K24 x OBLUFFS) - (K25 x CS)	ff
13.		CS	→	(K26 x OCORREL) + (K27 x OD)	ff
14.		BO	→	(K28 x CS) - (K29 x OH)	ff
15.		LAP	→	K30 - (K31 x BO)	ff
16.		SB	→	random(1,5)	ff
17.		MB	→	random(3,9)	ff
18.		BBS	→	random(10,15)	ff
19.		BB	→	random(8,14)	ff
20.		BBL	→	random(14,20)	ff
21.		H	→	VDHAND, VDHAND > 0	bf
22.		P	→	POT, POT > -1	bf
23.		B	→	LASTBET, 0 ≤ LASTBET < 21	bf
24.		BFO	→	BLUFFO, BLUFFO < 0 ∨ BLUFFO ≥ 0	bf

BLANK PAGE

25. PB → POTBET, POTBET $\geq$ 0 bf

26. R → ORP, -1 < ORP < 4 bf

27. OCS → OSTYLE, OSTYLE < 0 $\vee$ OSTYLE $\geq$ 0 bf

V. Values of Constants K1 Through K32

The values of the constants used in defining the production rules representing the heuristics for draw poker are given below.

K1 = 40
K2 = 22
K3 = 1
K4 = 9
K5 = 5
K6 = 30
K7 = 1
K8 = 6
K9 = 23
K10 = 7
K11 = 10
K12 = 1
K13 = 1
K14 = 21
K15 = 4
K16 = 20
K17 = 4
K18 = 27
K19 = 376
K20 = 10
K21 = 0
K22 = 6
K23 = .05
K24 = 1
K25 = 2
K26 = 1
K27 = 2
K28 = 8
K29 = 1
K30 = 5
K31 = 1
K32 = 8

APPENDIX C

SAMPLE OF GAMES PLAYED DURING PROFICIENCY TEST FOR BUILT-IN HEURISTICS

The following program output is from a game (5 hands) of draw poker played between the program and a human opponent via the Stanford PDP-6 timesharing system. This game is one of a five-game series used to test the proficiency of the program. The left column on each page is the series I game of the test, while the right column on each page is the corresponding series II game. The dialogue printed by the program starts at the left margin of each column, while the dialogue typed by the human opponent is indented five spaces.

The abbreviations used to represent playing cards are H: hearts, S: spades, C: clubs, and D: diamonds. Thus S8 is an eight of spades, D11 a jack of diamonds, and H14 an ace of hearts.

Note that each hand dealt the human player in series I (left column) is identical to the hand dealt the program in the corresponding r-o-p in series II (right column), and vice versa. Thus the hands held by the program in each r-o-p can be determined.

```
(REFEREE 5)


YOUR HAND IS     H5 H14 C12 H4 D9


I BET 1.
THE POT EQUALS 2.
YOUR BET ...

    1



I CALL
I REPLACE 3. CARDS
WHAT CARDS DO YOU WANT REPLACED ...

    (H5 H4 D9)

YOUR NEW CARDS ARE     C4 C14 H7



I BET 3.
THE POT EQUALS 6.
YOUR BET ...

    7



I DROP
YOU WIN
MY SCORE IS -6.
YOUR SCORE IS 6.



YOUR HAND IS     C4 H11 S7 S5 S8
THE POT EQUALS 2.
YOUR BET ...

    1
```

```
(REFEREE 5)


YOUR HAND IS     D2 S12 D6 D8 C7


I BET 2.
THE POT EQUALS 2.
YOUR BET ...

    CALL

I REPLACE 3. CARDS
WHAT CARDS DO YOU WANT REPLACED ...

    (D2 D6 C7)

YOUR NEW CARDS ARE     C6 C3 S4



I BET 7.
THE PCT EQUALS 6.
YOUR BET ...

    DROP

I WIN
MY SCORE IS 3.
YOUR SCORE IS A MERE -3.



YOUR HAND IS     C10 H12 D2 S10 S12
THE POT EQUALS 2.
YOUR BET ...

    5



I DROP
YOU WIN
MY SCORE IS 2.
YOUR SCORE IS A MERE -2.



YOUR HAND IS     D4 S3 D3 D7 D5
```

```
I BET 4.
THE POT EQUALS 4.
YOUR BET ...

    CALL

WHAT CARDS DO YOU WANT REPLACED ...

    (C4 S7 S5)

YOUR NEW CARDS ARE     S13 C5 H9
I REPLACE 1. CARD
THE POT EQUALS 12.
YOUR BET ...

    1



I BET 8.
THE POT EQUALS 14.
YOUR BET ...

    DROP

I WIN
MY SCORE IS 1.
YOUR SCORE IS A MERE -1.



YOUR HAND IS     D10 C2 S11 S14 D8



I BET 2.
THE POT EQUALS 2.
YOUR BET ...

    CALL

I REPLACE 3. CARDS
WHAT CARDS DO YOU WANT REPLACED ...

    (D10 C2 D8)

YOUR NEW CARDS ARE     H5 C8 C11



I BET 3.
THE POT EQUALS 6.
YOUR BET ...

    5
```

```
I BET 3.
THE POT EQUALS 2.
YOUR BET ...

    CALL

I REPLACE 3. CARDS
WHAT CARDS DO YOU WANT REPLACED ...

    (D4 D7 D5)

YOUR NEW CARDS ARE     C12 D9 C3



I BET 3.
THE POT EQUALS 8.
YOUR BET ...

    6



I CALL
MY HAND IS     S14 S11 H5 C8 C11
YOU WIN
MY SCORE IS -11.
YOUR SCORE IS 11.



YOUR HAND IS     C9 H6 H14 H7 H8
THE POT EQUALS 2.
YOUR BET ...

    3



I BET 2.
THE POT EQUALS 8.
YOUR BET ...

    CALL

WHAT CARDS DO YOU WANT REPLACED ...

    C9

YOUR NEW CARDS ARE     H4
I REPLACE 3. CARDS
THE POT EQUALS 12.
YOUR BET ...

    8
```

```
I BET 3.
THE POT EQUALS 22.
YOUR BET ...

    6


I BET 3.
THE POT EQUALS 40.
YOUR BET ...

    CALL

MY HAND IS    S3 D3 C12 D9 C3
I WIN
MY SCORE IS 24.
YOUR SCORE IS A MERE -24.


YOUR HAND IS    C14 S2 D6 D14 C7
THE POT EQUALS 2.
YOUR BET ...

    4


I CALL
WHAT CARDS DO YOU WANT REPLACED ...

    (S2 D6 C7)

YOUR NEW CARDS ARE    S6 D12 S4
I REPLACE 1. CARD
THE POT EQUALS 10.
YOUR BET ...

    5


I BET 3.
THE POT EQUALS 20.
YOUR BET ...

    2


I CALL
MY HAND IS    C14 D14 S6 D12 S4
YOU WIN
MY SCORE IS -25.
YOUR SCORE IS 25.


YOUR HAND IS    S3 C13 H4 H9 C14


I BET 3.
THE POT EQUALS 2.
YOUR BET ...

    CALL

I REPLACE 0. CARDS
WHAT CARDS DO YOU WANT REPLACED ...

    (S3 H4 H9)

YOUR NEW CARDS ARE    S2 D14 H14


I BET 18.
THE POT EQUALS 8.
YOUR BET ...

    20


I DROP
YOU WIN
MY SCORE IS -47.
YOUR SCORE IS 47.
YOU WIN THE GAME

NIL
```

```
I BET 3.
THE POT EQUALS 38.
YOUR BET ...

    CALL

MY HAND IS    H14 H9 H7 H6 H4
I WIN
MY SCORE IS 42.
YOUR SCORE IS A MERE -42.



YOUR HAND IS    H11 D3 C8 C7 D6



I BET 1.
THE POT EQUALS 2.
YOUR BET ...

    CALL

I REPLACE 3. CARDS
WHAT CARDS DO YOU WANT REPLACED ...

    (D3 C7 D6)

YOUR NEW CARDS ARE    D9 C4 C11



I BET 3.
THE POT EQUALS 4.
YOUR BET ...

    9



I BET 3.
THE POT EQUALS 28.
YOUR BET ...

    CALL

MY HAND IS    C14 C13 S2 D14 H14
I WIN
MY SCORE IS 59.
YOUR SCORE IS A MERE -59.
I WIN THE GAME

NIL
```

APPENDIX D

TRAINING TRIALS

FOR MANUAL-TRAINING HEURISTICS

The following program output is from the first 8 training trials given the learning program by a human trainer via the Stanford PDP-6 timesharing system. The dialogue printed by the program starts at the left margin, while the dialogue typed by the human trainer (who also acts as the program's opponent) is indented 5 spaces.

The abbreviations used to represent playing cards are H: hearts, S: spades, C: clubs, and D: diamonds. Thus S8 is an eight of spades, D11 a jack of diamonds, and H14 an ace of hearts.

Each time the program places a bet it first prints (1) the current csnumber, i.e., the number of the action rule which was used in making the bet decision, and (2) the hand it currently holds. The action rules are considered to be numbered from top to bottom, starting with 1.

```
(REFEREE 5)


YOUR HAND IS     S9 D2 H6 C9 D13


(CSNUMBER 1.)
(C3 H3 H13 H7 H2)

I BET 3.
OK?

    (BET SB BECAUSE YOUR HAND IS FAIR, AND THE POT IS SMALL)


(CSNUMBER 1.)
(C3 H3 H13 H7 H2)

I BET 1.
THE POT EQUALS 2.
YOUR BET ...

    5



(CSNUMBER 2.)
(C3 H3 H13 H7 H2)

I BET 3.
OK?

    (CALL BECAUSE YOUR HAND IS FAIR, AND THE LASTBET IS LARGE, AND
 ORP IS (EQUAL TO -1))


(CSNUMBER 2.)
(C3 H3 H13 H7 H2)

I CALL
I REPLACE 3. CARDS
WHAT CARDS DO YOU WANT REPLACED ...

    (D2 H6 D13)

YOUR NEW CARDS ARE     D5 C6 S10
```

```
(CSNUMBER 3.)
(C3 H3 S14 S12 H4)

I BET 1.
OK?

    (BET SB BECAUSE YOUR HAND IS FAIR, AND THE LASTBET IS SMALL)


(CSNUMBER 3.)
(C3 H3 S14 S12 H4)

I BET 1.
THE POT EQUALS 14.
YOUR BET ...

    4


(CSNUMBER 4.)
(C3 H3 S14 S12 H4)

I CALL
OK?

    (CALL BECAUSE YOUR HAND IS FAIR, THE POT IS LARGE, AND THE
 LASTBET IS LARGE)


(CSNUMBER 4.)
(C3 H3 S14 S12 H4)

I CALL
MY HAND IS     C3 H3 S14 S12 H4
YOU WIN
MY SCORE IS -12.
YOUR SCORE IS 12.


YOUR HAND IS     H8 D9 C8 H9 C14
THE POT EQUALS 2.
YOUR BET ...

    8
```

BLANK PAGE

```
(CSNUMBER 5.)
(H11 C7 S6 C4 D3)

I CALL
OK?

    (DROP BECAUSE YOUR HAND IS POOR, AND THE PBRATIO IS SMALL)


(CSNUMBER 5.)
(H11 C7 S6 C4 D3)

I DROP
YOU WIN
MY SCORE IS -13.
YOUR SCORE IS 13.


YOUR HAND IS     D14 S8 H14 D10 D4


(CSNUMBER 6.)
(S13 H12 C11 D8 S7)

I BET 15.
OK?

    (BET SB BECAUSE YOUR HAND IS POOR, THE POT IS SMALL, AND
 ORP IS (EQUAL TO -1))


(CSNUMBER 6.)
(S13 H12 C11 D8 S7)

I BET 1.
THE POT EQUALS 2.
YOUR BET ...

    10
```

```
(CSNUMBER 5.)
(S13 H12 C11 D8 S7)

I DROP
OK?

    (OK)


(CSNUMBER 5.)
(S13 H12 C11 D8 S7)

I DROP
YOU WIN
MY SCORE IS -15.
YOUR SCORE IS 15.


SHUFFLE



YOUR HAND IS     H6 H8 S10 C13 S3
THE POT EQUALS 2.
YOUR BET ...

    2



(CSNUMBER 6.)
(C8 C6 D5 H4 D3)

I BET 1.
OK?

    (CALL BECAUSE YOUR HAND IS POOR, THE POT IS LARGE, AND THE
 LASTE \LASTBET IS LARGE)


(CSNUMBER 6.)
(C8 C6 D5 H4 D3)
```

```
I CALL
WHAT CARDS DO YOU WANT REPLACED ...

    (H6 H8 S3)

YOUR NEW CARDS ARE     D7 D10 S8
I REPLACE 1. CARD
THE POT EQUALS 6.
YOUR BET ...

    6



(CSNUMBER 6.)
(C6 D5 H4 D3 S14)

I CALL
OK?

    (DROP BECAUSE YOUR HAND IS POOR, AND THE PBRATIO IS SMALL,
AND THE LASTBET IS LARGE)



(CSNUMBER 5.)
(C6 D5 H4 D3 S14)

I DROP
YOU WIN
MY SCORE IS -18.
YOUR SCORE IS 18.
```

APPENDIX E

SAMPLE OF GAMES PLAYED DURING PROFICIENCY TEST FOR MANUAL-TRAINING HEURISTICS

The following program output is from a game (5 hands) of draw poker played between the program and a human opponent via the Stanford PDP-6 timesharing system. This game is one of a five-game series used to test the proficiency of the program. The left column on each page is the series I game of the test, while the right column on each page is the corresponding series II game. The dialogue printed by the program starts at the left margin of each column, while the dialogue typed by the human opponent is indented five spaces.

The abbreviations used to represent playing cards are H: hearts, S: spades, C: clubs, and D: diamonds. Thus S8 is an eight of spades, D11 a jack of diamonds, and H14 an ace of hearts.

Note that each hand dealt the human player in series I (left column) is identical to the hand dealt the program in the corresponding r-o-p in series II (right column), and vice versa. Thus the hands held by the program in each r-o-p can be determined.

```
(REFEREE 5)


YOUR HAND IS     S7 H6 H10 D3 S10



I BET 2.
THE POT EQUALS 2.
YOUR BET ...

    1



I CALL
I REPLACE 3. CARDS
WHAT CARDS DO YOU WANT REPLACED ...

    (S7 H6 D3)

YOUR NEW CARDS ARE     D11 C10 S11



I BET 1.
THE POT EQUALS 8.
YOUR BET ...

    4



I CALL
MY HAND IS     S13 H9 D9 D4 D2
YOU WIN
MY SCORE IS -9.
YOUR SCORE IS 9.



YOUR HAND IS     D11 S11 H7 C11 C9
THE POT EQUALS 2.
YOUR BET ...

    3



I BET 2.
THE POT EQUALS 8.
YOUR BET ...

    CALL
```

```
(REFEREE 5)


YOUR HAND IS     S13 S5 D8 H9 S4



I BET 8.
THE POT EQUALS 2.
YOUR BET ...

    DROP

I WIN
MY SCORE IS 1.
YOUR SCORE IS A MERE -1.



YOUR HAND IS     H2 H13 D6 S6 H3
THE POT EQUALS 2.
YOUR BET ...

    3



I BET 2.
THE POT EQUALS 8.
YOUR BET ...

    CALL

WHAT CARDS DO YOU WANT REPLACED ...

    (H2 H13 H3)

YOUR NEW CARDS ARE     C3 C7 C2
I REPLACE 2. CARDS
THE POT EQUALS 12.
YOUR BET ...

    4



I BET 10.
THE POT EQUALS 20.
YOUR BET ...

    CALL

MY HAND IS     D11 S11 C11 S12 C14
I WIN
MY SCORE IS 21.
YOUR SCORE IS A MERE -21.
```

```
WHAT CARDS DO YOU WANT REPLACED ...

    (H7 C9)

YOUR NEW CARDS ARE     S12 C14
I REPLACE 3. CARDS
THE POT EQUALS 12.
YOUR BET ...

    9



I CALL
MY HAND IS     D6 S6 C3 C7 C2
YOU WIN
MY SCORE IS -24.
YOUR SCORE IS 24.



YOUR HAND IS     S2 S10 S13 D5 H10



I BET 11.
THE POT EQUALS 2.
YOUR BET ...

    CALL

I REPLACE 3. CARDS
WHAT CARDS DO YOU WANT REPLACED ...

    (S2 S13 D5)

YOUR NEW CARDS ARE     S3 C10 C4



I BET 4.
THE POT EQUALS 24.
YOUR BET ...

    4



I CALL
MY HAND IS     H14 S9 H12 D9 H11
YOU WIN
MY SCORE IS -44.
YOUR SCORE IS 44.
```

```
YOUR HAND IS     C8 D3 H4 H14 S9



I BET 4.
THE POT EQUALS 2.
YOUR BET ...

    CALL

I REPLACE 3. CARDS
WHAT CARDS DO YOU WANT REPLACED ...

    (C8 D3 H4)

YOUR NEW CARDS ARE     H12 D9 H11



I BET 3.
THE POT EQUALS 10.
YOUR BET ...

    6



I BET 7.
THE POT EQUALS 28.
YOUR BET ...

    CALL

MY HAND IS     S10 H10 S3 C10 C4
I WIN
MY SCORE IS 42.
YOUR SCORE IS A MERE -42.



YOUR HAND IS     S8 H5 H6 S14 D13
THE POT EQUALS 2.
YOUR BET ...

    2



I BET 3.
THE POT EQUALS 6.
YOUR BET ...

    CALL
```

```
YOUR HAND IS    S7 D12 S5 S4 C5
THE POT EQUALS 2.
YOUR BET ...

    4



I DROP
YOU WIN
MY SCORE IS -45.
YOUR SCORE IS 45.



YOUR HAND IS    H11 S8 S6 C8 D11



I BET 11.
THE POT EQUALS 2.
YOUR BET ...

    14



I DROP
YOU WIN
MY SCORE IS -57.
YOUR SCORE IS 57.
YOU WIN THE GAME

NIL
```

```
WHAT CARDS DO YOU WANT REPLACED ...

    (S8 H5 H6)

YOUR NEW CARDS ARE    C12 D14 H9
I REPLACE 3. CARDS
THE POT EQUALS 12.
YOUR BET ...

    5



I CALL
MY HAND IS    S5 C5 D7 C6 D2
YOU WIN
MY SCORE IS 31.
YOUR SCORE IS A MERE -31.



YOUR HAND IS    H7 H4 C14 S2 D6



I BET 1.
THE POT EQUALS 2.
YOUR BET ...

    CALL

I REPLACE 1. CARD
WHAT CARDS DO YOU WANT REPLACED ...

    (H4 S2 D6)

YOUR NEW CARDS ARE    H14 C4 H5



I BET 9.
THE POT EQUALS 4.
YOUR BET ...

    12



I CALL
MY HAND IS    H11 D11 S8 C8 H9
I WIN
MY SCORE IS 54.
YOUR SCORE IS A MERE -54.
I WIN THE GAME

NIL
```

APPENDIX F

SAMPLE OF GAMES PLAYED DURING PROFICIENCY TEST FOR BEFORE-TRAINING HEURISTICS

The following program output is from a game (5 hands) of draw poker played between the program and a human opponent via the Stanford PDP-6 timesharing system. This game is one of a five-game series used to test the proficiency of the program. The left column on each page is the series I game of the test, while the right column on each page is the corresponding series II game. The dialogue printed by the program starts at the left margin of each column, while the dialogue typed by the human opponent is indented five spaces.

The abbreviations used to represent playing cards are H: hearts, S: spades, C: clubs, and D: diamonds. Thus S8 is an eight of spades, D11 a jack of diamonds, and H14 an ace of hearts.

Note that each hand dealt the human player in series I (left column) is identical to the hand dealt the program in the corresponding r-o-p series II (right column), and vice versa. Thus the hands held by the program in each r-o-p can be determined.

```
(REFEREE 5)

YOUR HAND IS     D10 C10 D14 H10 S14


I BET 5.
THE POT EQUALS 2.
YOUR BIT ...

    9



I BET 8.
THE POT EQUALS 30.
YOUR BET ...

    11



I BET 17.
THE POT EQUALS 68.
YOUR BET ...

    20



I BET 19.
THE POT EQUALS 142.
YOUR BET ...

    CALL

I REPLACE 3. CARDS
WHAT CARDS DO YOU WANT REPLACED ...

    NONE



I BET 15.
THE POT EQUALS 180.
YOUR BET ...

    CALL

MY HAND IS     D13 S12 D8 C13 H4
YOU WIN
MY SCORE IS -185.
YOUR SCORE IS 185.
```

```
(REFEREE 5)

YOUR HAND IS     S12 C9 D13 D6 S4


I BET 20.
THE POT EQUALS 2.
YOUR BET ...

    DROP

I WIN
MY SCORE IS 1.
YOUR SCORE IS A MERE -1.



YOUR HAND IS     C14 S3 D7 H11 H6
THE POT EQUALS 2.
YOUR BET ...

    1



I BET 15.
THE POT EQUALS 4.
YOUR BET ...

    DROP

I WIN
MY SCORE IS 3.
YOUR SCORE IS A MERE -3.



YOUR HAND IS     H7 D9 C11 C6 S9


I BET 6.
THE POT EQUALS 2.
YOUR BET ...

    CALL

I REPLACE 3. CARDS
WHAT CARDS DO YOU WANT REPLACED ...

    (H7 C11 C6)

YOUR NEW CARDS ARE     C2 H5 H8
```

```
YOUR HAND IS     D5 H9 S13 S5 S6
THE POT EQUALS 2.
YOUR BET ...

    3



I BET 2.
THE POT EQUALS 8.
YOUR BET ...

    CALL

WHAT CARDS DO YOU WANT REPLACED ...

    (H9 S13  S6)

YOUR NEW CARDS ARE     C12 C3 C5
I REPLACE 3. CARDS
THE POT EQUALS 12.
YOUR BET ...

    11



I DROP
YOU WIN
MY SCORE IS -111.
YOUR SCORE IS 111.



YOUR HAND IS     C7 H13 S8 H3 H12



I BET 2.
THE POT EQUALS 2.
YOUR BET ...

    CALL

I REPLACE 3. CARDS
WHAT CARDS DO YOU WANT REPLACED ...

    (C7 S8 H3)

YOUR NEW CARDS ARE     H14 C8 D4



I DROP
YOU WIN
MY SCORE IS -114.
YOUR SCORE IS 114.
```

```
I BET 6.
THE POT EQUALS 14.
YOUR BET ...

    CALL

MY HAND IS     H13 H12 H14 C8 D4
YOU WIN
MY SCORE IS -18.
YOUR SCORE IS 18.



YOUR HAND IS     C11 H3 D3 S7 H10
THE POT EQUALS 2.
YOUR BET ...

    3



I CALL
WHAT CARDS DO YOU WANT REPLACED ...

    (C11 S7 H10)

YOUR NEW CARDS ARE     C12 C14 D4
I REPLACE 1. CARD
THE POT EQUALS 8.
YOUR BET ...

    4



I BET 4.
THE POT EQUALS 16.
YOUR BET ...

    CALL

MY HAND IS     H11 D10 H8 D7 H12
YOU WIN
MY SCORE IS -22.
YOUR SCORE IS 22.



YOUR HAND IS     H2 H13 C10 S10 S13



I BET 6.
THE POT EQUALS 2.
YOUR BET ...

    5
```

```
YOUR HAND IS    H8 D10 D5 H11 D7
THE POT EQUALS 2.
YOUR BET ...

    1

                                                    I DROP
                                                    YOU WIN
                                                    MY SCORE IS -29.
                                                    YOUR SCORE IS 29.
                                                    YOU WIN THE GAME

                                                    NIL

I BET 10.
THE POT EQUALS 4.
YOUR BET ...

    DROP

I WIN
MY SCORE IS -112.
YOUR SCORE IS 112.


YOUR HAND IS    S3 S2 C13 H5 S8


I BET 3.
THE POT EQUALS 2.
YOUR BET ...

    CALL

I REPLACE 1. CARD
WHAT CARDS DO YOU WANT REPLACED ...

    (S3 S2 H5)

YOUR NEW CARDS ARE    S14 D12 C5


I DROP
YOU WIN
MY SCORE IS -116.
YOUR SCORE IS 116.
YOU WIN THE GAME

NIL
```

APPENDIX G

TRAINING TRIALS

FOR AUTOMATIC-TRAINING HEURISTICS

The following program output is from training trials 6 through 10 given the learning program by a program trainer via the Stanford PDP-6 timesharing system. The dialogue printed by the program being trained and by the program trainer starts at the left margin, while the dialogue typed by the human opponent is indented 5 spaces.

The abbreviations used to represent playing cards are H: hearts, S: spades, C: clubs, and D: diamonds. Thus S8 is an eight of spades, D11 a jack of diamonds, and H14 an ace of hearts.

Each time the program places a bet it first prints (1) the current csnumber, i.e., the number of the action rule which was used in making the bet decision, and (2) the hand it currently holds. The action rules are considered to be numbered from top to bottom, starting with 1.

```
(REFEREE 5)

SHUFFLE



YOUR HAND IS     S10 H3 H5 J14 H14


(CSNUMBER 2.)
(D2 C2 H11 S9 D4)

I BET 4.
OK?


(CSNUMBER 19.)
(D2 C2 H11 S9 D4)


(OK)


(CSNUMBER 2.)
(D2 C2 H11 S9 D4)

I BET 2.
THE POT EQUALS 2.
YOUR BET ...

    6



(CSNUMBER 1.)
(D2 C2 H11 S9 D4)

I CALL
OK?


(CSNUMBER 18.)
(D2 C2 H11 S9 D4)


(OK)


(CSNUMBER 1.)
(D2 C2 H11 S9 D4)
```

```
(CSNUMBER 1.)
(D2 C2 H11 S9 D4)

I CALL
I REPLACE 3. CARDS
WHAT CARDS DO YOU WANT REPLACED ...

    (S10 H3 H5)

YOUR NEW CARDS ARE     C7 C4 C10


(CSNUMBER 2.)
(D2 C2 D13 S5 S4)

I BET 3.
OK?


(CSNUMBER 19.)
(D2 C2 D13 S5 S4)


(OK)


(CSNUMBER 2.)
(D2 C2 D13 S5 S4)

I BET 3.
THE POT EQUALS  8.
YOUR BET ...

    8


(CSNUMBER 1.)
(D2 C2 D13 S5 S4)

I CALL
OK?


(CSNUMBER 18.)
(D2 C2 D13 S5 S4)


(OK)


(CSNUMBER 1.)
(D2 C2 D13 S5 S4)
```

```
I CALL
MY EAND IS     D2 C2 D13 S5 S4
YOU WIN
MY SCORE IS -20.
YOUR SCORE IS 20.



YOUR HAND IS     D14 S11 H6 S12 S13
THE POT EQUALS 2.
YOUR BET ...

    6



(CSNUMBER 5.)
(C13 C11 D8 D3 S2)

I CALL.
OK?


(CSNUMBER 28.)
(C13 C11 D8 D3 S2)


(DROP BECAUSE THE HAND IS POOR)


(CSNUMBER 5.)
(C13 C11 D8 D3 S2)

I DROP
YOU WIN
MY SCORE IS -21.
YOUR SCORE IS 21.



YOUR HAND I      D12 C6 H2 D11 S7


(CSNUMBER 3.)
(C12 H12 H8 S6 S3)

I BET 7.
OK?


(CSNUMBER 12.)
(C12 H12 H8 S6 S3)
```

```
(OK)


(CSNUMBER 3.)
(C12 H12 H8 S6 S3)

I BET 7.
THE POT EQUALS 2.
YOUR BET ...

    DROP

I WIN
MY SCORE IS -20.
YOUR SCORE IS 20.



YOUR HAND IS     C9 H7 H9 D10 H4
THE POT EQUALS 2.
YOUR BET ...

    4



(CSNUMBER 3.)
(C8 S8 H10 D7 D6)

I BET 5.
OK?


(CSNUMBER 18.)
(C8 S8 H10 D7 D6)


(CALL BECAUSE THE HAND IS FAIR THE LASTBET IS LARGE)


(CSNUMBER 1.)
(C8 S8 H10 D7 D6)

I CALL
WHAT CARDS DO YOU WANT REPLACED ...

    (H7 D10 H4)

YOUR NEW CARDS ARE     D9 C14 C5
I REPLACE 3. CARDS


SHUFFLE
THE POT EQUALS 10.
YOUR BET ...

    8
```

```
(CSNUMBER 1.)
(C8 S8 H13 D5 C3)

I CALL
OK?


(CSNUMBER 18.)
(C8 S8 H13 D5 C3)


(OK)


(CSNUMBER 1.)
(C8 S8 H13 D5 C3)


I CALL
MY HAND IS    C8 S8 H13 D5 C3
YOU WIN
MY SCORE IS -33.
YOUR SCORE IS 33.



YOUR HAND IS    H11 S13 H5 C10 H6


(CSNUMBER 3.)
(C12 D12 C11 S5 H4)

I BET 7.
OK?


(CSNUMBER 9.)
(C12 D12 C11 S5 H4)


(BET SB BECAUSE THE HAND IS GOOD THE BLUFFS IS LARGE THE ORP IS (EQUA
L TO -1.))


(CSNUMBER 3.)
(C12 D12 C11 S5 H4)

I BET 3.
THE POT EQUALS 2.
YOUR BET ...

    CALL
```

```
    CALL

I REPLACE 3. CARDS
WHAT CARDS DO YOU WANT REPLACED ...

    (H5 H6)

YOUR NEW CARDS ARE     D14 C14


(CSNUMBER 4.)
(C12 D12 S3 D5 D13)

I BET 6.
OK?


(CSNUMBER 16.)
(C12 D12 S3 D5 D13)


(BET BB BECAUSE THE HAND IS FAIR THE POT IS SMALL THE LASTBET IS SMAL
L THE BLUFFS IS LARGE)


(CSNUMBER 2.)
(C12 D12 S3 D5 D13)

I BET 2.
THE POT EQUALS 8.
YOUR BET ...

    10


(CSNUMBER 1.)
(C12 D12 S3 D5 D13)

I CALL
OK?


(CSNUMBER 17.)
(C12 D12 S3 D5 D13)


(DROP BECAUSE THE HAND IS FAIR THE LASTBET IS LARGE THE PBRATIO IS SM
ALL THE ORP IS (NOT (EQUAL TO -1.)))
```

```
(CSNUMBER 1.)
(C12 D12 S3 D5 D13)

I DROP
YOU WIN
MY SCORE IS -39.
YOUR SCORE IS 39.
YOU WIN THE GAME

NIP
```

APPENDIX H

SAMPLE OF GAMES PLAYED DURING PROFICIENCY TEST FOR AUTOMATIC-TRAINING HEURISTICS

The following program output is from a game (5 hands) of draw poker played between the program and a human opponent via the Stanford PDP-6 timesharing system. This game is one of a five-game series used to test the proficiency of the program. The left column on each page is the series I game of the test, while the right column on each page is the corresponding series II game. The dialogue printed by the program starts at the left margin of each column, while the dialogue typed by the human opponent is indented five spaces.

The abbreviations used to represent playing cards are H: hearts, S: spades, C: clubs, and D: diamonds. Thus S8 is an eight of spaces, D11 a jack of diamonds, and H14 an ace of hearts.

Note that each hand dealt the human player in series I (left column) is identical to the hand dealt the program in the corresponding r-o-p in series II (right column), vice versa. Thus the hands held by the program in each r-o-p can be determined.

```
(REFEREE 5)

YOUR HAND IS    H13 S14 D14 S12 D10


I BET 4.
THE POT EQUALS 2.
YOUR BET ...

    CALL

I REPLACE 3. CARDS
WHAT CARDS DO YOU WANT REPLACED ...

    (H13 S12 D10)

YOUR NEW CARDS ARE    H14 C6 D13


I BET 4.
THE POT EQUALS 10.
YOUR BET ...

    7



I DROP
YOU WIN
MY SCORE IS -9.
YOUR SCORE IS 9.



YOUR HAND IS    D12 C5 S10 D14 H3
THE POT EQUALS 2.
YOUR BET ...

    3



I CALL
WHAT CARDS DO YOU WANT REPLACED ...

    (C5 S10 H3)

YOUR NEW CARDS ARE    H10 D2 H6
I REPLACE 3. CARDS
THE POT EQUALS 8.
YOUR BET ...

    4
```

```
(REFEREE 5)

YOUR HAND IS    C13 C10 S11 S5 C2


I BET 8.
THE POT EQUALS 2.
YOUR BET ...

    CALL

I REPLACE 3. CARDS
WHAT CARDS DO YOU WANT REPLACED ...

    (C10 S5 C2)

YOUR NEW CARDS ARE    D8 H5 H7


I BET 3.
THE POT EQUALS 18.
YOUR BET ...

    CALL

MY HAND IS    S14 D14 H14 C6 D13
I WIN
MY SCORE IS 12.
YOUR SCORE IS A MERE -12.



YOUR HAND IS    D9 C7 H4 S13 D6
THE POT EQUALS 2.
YOUR BET ...

    1



I CALL
WHAT CARDS DO YOU WANT REPLACED ...

    (C7 H4 D6)

YOUR NEW CARDS ARE    S8 S5 S11
I REPLACE 3. CARDS
THE POT EQUALS 4.
YOUR BET ...

    1
```

```
I DROP
YOU WIN
MY SCORE IS -13.
YOUR SCORE IS 13.


YOUR HAND IS    S4 D8 C9 D11 C12


I BET 2.
THE POT EQUALS 2.
YOUR BET ...

    CALL

I REPLACE 3. CARDS
WHAT CARDS DO YOU WANT REPLACED ...

    (S4 D8 C9)

YOUR NEW CARDS ARE    S12 H6 C3


I BET 5.
THE POT EQUALS 6.
YOUR BET ...

    11


I BET 3.
THE POT EQUALS 38.
YOUR BET ...

    CALL

MY HAND IS    H13 H12 C11 H9 H11
YOU WIN
MY SCORE IS -35.
YOUR SCORE IS 35.


YOUR HAND IS    D3 D13 C8 D4 H5
THE POT EQUALS 2.
YOUR BET ...

    2
```

```
I CALL
MY HAND IS    D14 D12 H10 D2 H6
I WIN
MY SCORE IS 15.
YOUR SCORE IS A MERE -15.


YOUR HAND IS    H13 S2 H12 C4 S9


I BET 2.
THE POT EQUALS 2.
YOUR BET ...

    CALL

I REPLACE 3. CARDS
WHAT CARDS DO YOU WANT REPLACED ...

    (S2 C4 S9)

YOUR NEW CARDS ARE    C11 H9 H11


I BET 7.
THE POT EQUALS 6.
YOUR BET ...

    4


I BET 8.
THE POT EQUALS 26.
YOUR BET ...

    CALL

MY HAND IS    C12 D11 S13 H8 C3
I WIN
MY SCORE IS 37.
YOUR SCORE IS A MERE -37.


YOUR HAND IS    D10 C2 C13 H7 H2
THE POT EQUALS 2.
YOUR BET ...

    3
```

```
I CALL
WHAT CARDS DO YOU WANT REPLACED ...

    (D3 D4 H5)

YOUR NEW CARDS ARE     C10 D5 D7
I REPLACE 3. CARDS
TSE POT EQUALS 6.
YOUR BET ...

    4


I CALL
MY HAND IS     C2 H2 H14 S6 S3
I WIN
MY SCORE IS -28.
YOUR SCORE IS 28.


YOUR HAND IS     S3 D6 H4 H3 H14


I BET 4.
THE POT EQUALS 2.
YOUR BET ...

    1


I CALL
I REPLACE 1. CARD
WHAT CARDS DO YOU WANT REPLACED ...

    (D6 H4 H14)

YOUR NEW CARDS ARE     S7 S11 C9


I BET 5.
THE POT EQUALS 12.
YOUR BET ...

    CALL

MY HAND IS     C14 S13 D12 D11 H13
I WIN
MY SCORE IS -17.
YOUR SCORE IS 17.
YOU WIN THE GAME

NIL
```

```
I CALL
WHAT CARDS DO YOU WANT REPLACED ...

    (D10 C13 H7)

YOUR NEW CARDS ARE     H14 S6 S3
I REPLACE 3. CARDS
THE POT EQUALS 8.
YOUR BET ...

    4


I DROP
YOU WIN
MY SCORE IS 33.
YOUR SCORE IS A MERE -33.


YOUR HAND IS     D11 S13 C4 C14 D12


I BET 3.
THE POT EQUALS 2.
YOUR BET ...

    1


I CALL
I REPLACE 3. CARDS
WHAT CARDS DO YOU WANT REPLACED ...

    C4

YOUR NEW CARDS ARE     H13


I BET 1.
THE POT EQUALS 10.
YOUR BET ...

    4


I CALL
MY HAND IS     S3 H3 S7 S11 C9
YOU WIN
MY SCORE IS 23.
YOUR SCORE IS A MERE -23.
I WIN THE GAME

NIL
```

APPENDIX I

LOGICAL STATEMENTS FOR DRAW POKER

I. Rules and Axioms for Draw Poker

The rules and axioms for draw poker used by the computer program are listed below. In these statements "action" refers to the decision made by the program while "oppaction" refers to the decision made by the program's opponent. A low bet is defined as a bet from 1 to 9 , while a high bet is defined as one from 10 to 20 .

Poker Rules:

1. action(call) ∧ higher(yourhand,opphand) ⊃ add(lastbet,pot) ∧ add(pot,yourscore)
2. oppaction(call) ∧ higher(yourhand,opphand) ⊃ add(lastbet,pot) ∧ add(pot,yourscore)
3. action(call) ∧ higher(opphand,yourhand) ⊃ add(lastbet,pot) ∧ sub(pot,yourscore)
4. oppaction(call) ∧ higher(opphand,yourhand) ⊃ add(lastbet,pot) ∧ add(pot,yourscore)
5. action(drop) ⊃ sub(pot,yourscore)
6. oppaction(drop) ⊃ add(pot,yourscore)
7. action(bet low) ⊃ add(lastbet,pot)
8. action(bet high) ⊃ add(lastbet,pot)
9. oppaction(bet low) ⊃ add(lastbet,pot)
10. oppaction(bet high) ⊃ add(lastbet,pot)

Poker Axioms:

1. action(drop) ⊃ keepsmall(pot)
2. action(call) ⊃ unsureofhand(you)
3. onlycalled(opp) ⊃ unsureofhand(opp)
4. action(bet low) ∨ action(bet high) ⊃ keepsbetting(you)
5. oppaction(bet low) ∨ oppaction(bet high) ⊃ keepsbetting(opp)
6. keepsbetting(opp) ∧ keepsbetting(you) ⊃ buildup(pot)
7. action(bet high) ∧ higher(opphand,yourhand) ⊃ bluffed(opp)
8. goodhand(x) ∧ didbet(x) ⊃ surehandwillwin(x)
9. unsureofhand(you) ∧ seemsureofhand(opp) ⊃ makelargenough(pot)
10. pot(large) ∨ lastbetopp(bet high) ⊃ seemsureofhand(opp)
11. (action(call) ∨ action(bet low) ∨ action(bet high)) ∧ higher(yourhand,opphand) ⊃ eventually(add(pot,yourscore))
12. bad(opphand) ∧ bluffed(opp) ∧ notprevoppaction(bet high) ⊃ prob(oppaction(drop))
13. (action(bet high) ∨ action(bet low)) ∧ surehandwillwin(opp) ⊃ prob(oppaction(bet low)) ∧ prob(oppaction(bet high))

14. action(bet low) ∧ good(opphand) ∧ unsureofhand(opp) ⊃
prob(oppaction(bet low)) ∧ prob(oppaction(call))
15. action(bet low) ∧ bad(opphand) ⊃ prob(oppaction(bet low)) ∧ prob(oppaction(call))

General Axioms:

1. x ⊃ eventually(x)
2. (buildup(x) ∨ makelargenough(x)) ∧ eventually(add(x,z))
   ∨ add(x,z)
   ∨ (keepsmall(x) ∧ sub(x,z) ⊃ maximize(z)

The meanings of the predicates shown above tend to be self-evident, however the logical statements are written out in detail in Appendix I, Part II.

II. Description of Rules and Axioms for Draw Poker

The rules and axioms for draw poker listed in Appendix I, Part I are described in detail below.

Poker Rules:

1. If you or your opponent calls, and your hand is higher than your opponent's hand then the last bet is added to the pot, after which the pot is added to yourscore.
2. If you or your opponent calls and your opponent's hand is higher than your hand, then the last bet is added to the pot, after which the pot is subtracted from your score.
3. If you drop, then the pot is subtracted from your score.
4. If your opponent drops, then the pot is added to your score.
5. If you or your opponent bets, then that bet is added to the pot.

Poker Axioms:

1. If you drop, then you keep the pot small.
2. If you call, you are unsure your hand will win.
3. If your opponent calls but does not bet in an r-o-p, then he is unsure his hand will win.
4. If you bet, then you have kept the betting going.
5. If your opponent bets, then he has kept the betting going.
6. If both you and your opponent keep the betting going, then the amount of money in the pot builds up.
7. If you bet high and your opponent's hand is higher than your hand, the you have bluffed.

8. If a player has a good hand and has just bet, then he is sure that his hand will win.
9. If you are unsure your hand will win and the opponent seems sure his hand will win, then you have made the pot large enough.
10. If the pot is large or the last bet by the opponent was large, then the opponent seems sure his hand will win.
11. If you call or bet and your hand is higher than your opponent's hand, then you will eventually add the pot to your score.
12. If your opponent has a bad hand and you bluff but have not previously bet high in the present r-o-p, then it is probable that your opponent will drop.
13. If you bet and your opponent is sure that his hand will win, then it is probable that your opponent will also bet.
14. If you bet low and your opponent has a good hand and is unsure his hand will win, then it is probable that your opponent will bet low or call.
15. If you bet low and your opponent has a bad hand, then it is probable that your opponent will bet low or call.

General Axioms:

1. If x is now true then x will be true in the future, that is eventually. (Here x must be a member of a class of predicates whose values are irreversible within the time limit under consideration.)
2. If you increase the size of x or make x large enough and eventually add x to z, or if you just add x to z, or if you keep x small and subtract x from z then you tend to maximize z.

III. Example of Deduction Procedure Using Rules and Axioms for Draw Poker

Assume the predicates in the logical statements are set as follows:

higher(yourhand,opphand) = F	higher(opphand,yourhand) = T
notprevoppaction(bet high) = T	lastbetopp(bet high) = F
onlycalled(opp) = T	pot(large) = F
goodhand(you) = F	goodhand(opp) = F
good(opphand) = F	bad(opphand) = T
didbet(you) = T	didbet(opp) = F

In this case maximize(yourscore) matches maximize(z) in the right side of the last logical statement when "yourscore" is substituted for z. Thus the program tries to make the left side of this statement true, which is the expression:

(buildup(x) ∨ (keepsmall(x) ∧ eventually(add(x,yourscore))

∨ add(x,yourscore) ∨ (keepsmall(x) ∧ sub(x,yourscore)) .

This expression has the form a ∨ b ∨ c , so the program first attempts to make a true. If this fails it tries to make b true, and if this also fails it tries c . Here a has the form $a_1 \wedge a_2$; accordingly both a_1 and a_2 must be made true if a is to be true. But a_2 = eventually(add(x,yourscore)) which matches only the right side of axiom 11 of the poker axioms. For a_2 to be true, the left part of axiom 11 must be true, but this is false since higher(yourhand,opphand) is false. Consequently, it cannot be shown that a_2 can be made true, or that a can be made true.

Now the program attempts to make b true, where b = add(x,yourscore). This expression matches the right sides of poker rules 1, 2, and 6 (b is considered a match for a ∧ b since if it is shown that a ∧ b is true it is also shown that b is true), but the left sides of rules 1 and 2 cannot be made true since they both contain higher(yourhand,opphand), which is false.

However, the right side of rule 6 can be made true if oppaction(drop) can be made true. This expression matches only the right side of poker axiom 12 and will be true if the left side of axiom 12, bad(opphand) ∧ bluffed(opp) ∧ notprevoppaction(bet high), can be made true. But bad(opphand) and notprevoppaction(bet high) are both prediactes set to true by the program, so the right side of axiom 12 is true if bluffed(opp) can be made true. This expression matches only the right side of poker axiom 7 and is true if the left side of axiom 7, action(bet high) ∧ higher(opphand,yourhand), can

be made true. Since higher(opphand,yourhand) is one of the predicates initially set to true by the program, bluffed(opp) is true if action(bet high) can be made true. But his can be made true by having the program make the decision to bet high; thus the decision to bet high makes bluffed(opp), prob(oppaction(drop)), add(pot,yourscore), and maximize(yourscore) all true. As a consequence, the program deduces that it should have bet high in the given situation in order to have maximized its score.

BLANK PAGE

APPENDIX J

TRAINING TRIALS

FOR IMPLICIT-TRAINING HEURISTICS

The following program output is from the first 5 learning trials given the learning program via the Stanford PDP-6 timesharing system. The dialogue printed by the program starts at the left margin, while the dialogue typed by the human opponent is indented 5 spaces.

The abbreviations used to represent playing cards are H: hearts, S: spades, C: clubs, and D: diamonds. Thus S8 is an eight of spades, D11 a jack of diamonds, and H14 an ace of hearts.

Each time the program places a bet it first prints (1) the current csnumber, i.e., the number of the action rules which was used in making the bet decision, and (2) the hand it currently holds. The action rules are considered to be numbered from top to bottom, starting with 1.

At the end of each r-o-p the program prints the following for each bet decision it makes after cards are replaced: (1) the csnumber for that bet decision, (2) a list of acceptable bet decisions, (3) and (4) the decision chosen from the list of acceptable ones, which is inserted in the action rule list as an action rule, and (5) the program subvector existing at the time it made the bet decision, together with the bet decisions made by the program and the opponent.

```
(REFEREE 5   )


  YOUR HAND IS     H4 D10 C10 C9 D14


  (CSNUMBER 1.)
  (H6 D6 H14 D9 D5)

  I BET 1.
  THE POT EQUALS 2.
  YOUR BET ...

      CALL

  I REPLACE 3. CARDS
  WHAT CARDS DO YOU WANT REPLACED ...

      (H4 C9 D14)

  YOUR NEW CARDS ARE     D7 S11 D12


  (CSNUMBER 1.)
  (H6 D6 S6 D8 H10)

  I BET 1.
  THE POT EQUALS 4.
  YOUR BET ...

      2



  (CSNUMBER 1.)
  (H6 D6 S6 D8 H10)

  I BET 11.
  THE POT EQUALS 10.
  YOUR BET ...

      CALL

  MY HAND IS     H6 D6 S6 D8 H10
```

```
(CSNUMBER 1.)
(BETHIGH BETLOW)
BETLOW
(BET SSS)


(CSNUMBER 2.)
(BETHIGH BETLOW)
BETLOW
(BET SSS)
(((52. 4. 0. -6. 4. 3. 0.) BETLOW BETLOW) ((52. 6. 2. -6. 3. 3. 0.) B
ETHIGH CALL)) I WIN
MY SCORE IS 16.
YOUR SCORE IS A MERE -16.


SHUFFLE



YOUR HAND IS     S12 D3 S9 C7 D12
THE POT EQUALS 2.
YOUR BET ...

    7



(CSNUMBER 2.)
(S4 D4 H13 H12 H11)

I BET 13.
THE POT EQUALS 16.
YOUR BET ...

    CALL

WHAT CARDS DO YOU WANT REPLACED ...

    (D3 S9 C7)

YOUR NEW CARDS ARE     D7 S10 H9
I REPLACE 3. CARDS
THE POT EQUALS 42.
YOUR BET ...

    5
```

```
(CSNUMBER 2.)
(S4 D4 S11 D8 C2)

I CALL
MY HAND IS     S4 D4 S11 D8 C2


(CSNUMBER 2.)
(CALL)
CALL
CALL
(((10. 42. 5. -6. 8. 3. 0.) CALL NIL)) YOU WIN
MY SCORE IS -10.
YOUR SCORE IS 10.



YOUR HAND IS     S14 H6 D5 D6 H4


(CSNUMBER 3.)
(D14 S13 S8 S5 H2)

I BET 6.
THE POT EQUALS 2.
YOU
    BET ...

    CALL

I REPLACE 3. CARDS
WHAT CARDS DO YOU WANT REPLACED ...

    (D5 H4)

YOUR NEW CARDS ARE     D2 H8


(CSNUMBER 3.)
(D14 S13 C8 C9 C11)

I BET 5.
THE POT EQUALS 14.
YOUR BET ...

    CALL

MY HAND IS     D14 S13 C8 C9 C11
```

```
(CSNUMBER 3.)
(DROP)
DROP
DROP
(((2. 14. 0. -14. 14. 2. -1.) BETLOW CALL)) YOU WIN
MY SCORE IS -22.
YOUR SCORE IS 22.



YOUR HAND IS     C3 H14 C6 C4 H3
THE POT EQUALS 2.
YOUR BET ...

    2



(CSNUMBER 4.)
(S7 H7 C10 D9 S3)

I CALL
WHAT CARDS DO YOU WANT REPLACED ...

    (C6 C4)

YOUR NEW CARDS ARE     C5 H10
I REPLACE 3. CARDS
THE POT EQUALS 6.
YOUR BET ...

    3



(CSNUMBER 4.)
(S7 H7 D10 D13 C12)

I CALL
MY HAND IS     S7 H7 D10 D13 C12


(CSNUMBER 4.)
(BETLOW)
BETLOW
(BET SSS)
(((13. 6. 3. -14. 2. 2. -1.) CALL NIL)) I WIN
MY SCORE IS -16.
YOUR SCORE IS 16.
```

APPENDIX K

SAMPLE OF GAMES PLAYED DURING PROFICIENCY TEST FOR IMPLICIT-TRAINING HEURISTICS

The following program output is from a game (5 hands) of draw poker played between the program and a human opponent via the Stanford PDP-6 timesharing system. This game is one of a five-game series used to test the proficiency of the program. The left column on each page is the series I game of the test, while the right column on each page is the corresponding series II game. The dialogue printed by the program starts at the left margin of each column, while the dialogue typed by the human opponent is indented five spaces.

The abbreviations used to represent playing cards are H: hearts, S: spades, C: clubs, and D: diamonds. Thus S8 is an eight of spades, D11 a jack of diamonds, and H14 an ace of hearts.

Note that each hand dealt the human player in series I (left column) is identical to the hand dealt the program in the corresponding r-o-p in series II (right column), and vice versa. Thus the hands held by the program in each r-o-p can be determined.

BLANK PAGE

```
(REFEREE 5)


YOUR HAND IS     D6 D13 C12 S14 S3


I BET 3.
THE POT EQUALS 2.
YOUR BET ...

    CALL

I REPLACE 3. CARDS
WHAT CARDS DO YOU WANT REPLACED ...

    (D6 S3 C12)

YOUR NEW CARDS ARE     C13 C11 C3


I BET 8.
THE POT EQUALS 8.
YOUR BET ...

    8


I CALL
MY HAND IS     S8 D8 H7 D3 D5
YOU WIN
MY SCORE IS -20.
YOUR SCORE IS 20.


YOUR HAND IS     C14 D4 H11 H13 S7
THE POT EQUALS 2.
YOUR BET ...

    3


I BET 7.
THE POT EQUALS 8.
YOUR BET ...

    CALL
```

```
(REFEREE 5)


YOUR HAND IS     H14 S8 C6 D8 S4


I BET 7.
THE POT EQUALS 2.
YOUR BET ...

    CALL

I REPLACE 3. CARDS
WHAT CARDS DO YOU WANT REPLACED ...

    (H14 C6 S4)

YOUR NEW CARDS ARE     H7 D3 D5


I BET 5.
THE POT EQUALS 16.
YOUR BET ...

    3


I BET 1.
THE POT EQUALS 32.
YOUR BET ...

    2


I BET 1.
THE POT EQUALS 38.
YOUR BET ...

    CALL

MY HAND IS     S14 D13 C13 C11 C3
I WIN
MY SCORE IS 20.
YOUR SCORE IS A MERE -20.
```

```
WHAT CARDS DO YOU WANT REPLACED ...

    (D4 H11 S7)

YOUR NEW CARDS ARE     D9 D7 C2
I REPLACE 3. CARDS
THE POT EQUALS 22.
YOUR BET ...

    2



I BET 2.
THE POT EQUALS 26.
YOUR BET ...

    CALL

MY HAND IS     D12 C9 S9 H5 H2
I WIN
MY SCORE IS -5.
YOUR SCORE IS 5.


YOUR HAND IS     D10 S13 C8 S2 S5



I BET 7.
THE POT EQUALS 2.
YOUR BET ...

    CALL

I REPLACE 3. CARDS
WHAT CARDS DO YOU WANT REPLACED ...

    (C8 S2 S5)

YOUR NEW CARDS ARE     C7 H10 S10



I BET 6.
THE POT EQUALS 16.
YOUR BET ...

    14
```

```
YOUR HAND IS     D2 H8 D12 C9 C5
THE POT EQUALS 2.
YOUR BET ...

    3



I BET 6.
THE POT EQUALS 8.
YOUR BET ...

    CALL

WHAT CARDS DO YOU WANT REPLACED ...

    (D2 H8 C5)

YOUR NEW CARDS ARE     S9 H5 H2
I REPLACE 3. CARDS
THE POT EQUALS 20.
YOUR BET ...

    4



I DROP
YOU WIN
MY SCORE IS 10.
YOUR SCORE IS A MERE -10.


YOUR HAND IS     S11 D14 H4 H3 S12



I BET 3.
HE POT EQUALS 2.
YOUR BET ...

    CALL

I REPLACE 3. CARDS
WHAT CARDS DO YOU WANT REPLACED ...

    (S11 H4 H3)

YOUR NEW CARDS ARE     H6 H12 S6
```

```
I BET 16.
THE POT EQUALS 56.
YOUR BET ...

    CALL

MY HAND IS     D14 S12 H6 H12 S6
YOU WIN
MY SCORE IS -51.
YOUR SCORE IS 51.



YOUR HAND IS     C11 D2 H10 S2 C7
THE POT EQUALS 2.
YOUR BET ...

    2



I BET 14.
THE POT EQUALS 6.
YOUR BET ...

    CALL

WHAT CARDS DO YOU WANT REPLACED ...

    (C11 H10 C7)

YOUR NEW CARDS ARE     S5 C13 D11
I REPLACE 3. CARDS
THE POT EQUALS 34.
YOUR BET ...

    1



I BET 1.
THE POT EQUALS 36.
YOUR BET ...

    CALL

MY HAND IS     H12 S12 H14 C10 D9
I WIN
MY SCORE IS -32.
YOUR SCORE IS 32.



YOUR HAND IS     C2 D14 H8 H13 S4
```

```
I BET 7.
THE POT EQUALS 8.
YOUR BET ...

    13



I BET 12.
THE POT EQUALS 48.
YOUR BET ...

    CALL

MY HAND IS     S13 D10 C7 H10 S10
I WIN
MY SCORE IS 46.
YOUR SCORE IS A MERE -46.



YOUR HAND IS     H12 H2 S9 D6 S12
THE POT EQUALS 2.
YOUR BET ...

    6



I CALL
WHAT CARDS DO YOU WANT REPLACED ...

    (H2 S9 D6)

YOUR NEW CARDS ARE     H14 C10 D9
I REPLACE 3. CARDS
THE POT EQUALS 14.
YOUR BET ...

    5



I CALL
MY HAND IS     D2 S2 S5 C13 D11
YOU WIN
MY SCORE IS 34.
YOUR SCORE IS A MERE -34.



YOUR HAND IS Y   C6 C8 S3 S8 H3
```

```
YOUR HAND IS     C2 D14 H8 H13 S4


I BET 7.
THE POT EQUALS 2.
YOUR BET ...

    CALL

I REPLACE 1. CARD
WHAT CARDS DO YOU WANT REPLACED ...

    (C2 H8 S4)

YOUR NEW CARDS ARE     H11 S13 S7


I BET 8.
THE POT EQUALS 16.
YOUR BET ...

    9


I BET 4.
THE POT EQUALS 50.
YOUR BET ...

    CALL

MY HAND IS     C8 S8 S3 H3 S6
I WIN
MY SCORE IS -3.
YOUR SCORE IS 3.
YOU WIN THE GAME

NIL
```

```
YOUR HAND IS Y  C6 C8 S3 S8 H3


I BET 7.
THE POT EQUALS 2.
YOUR BET ...

    CALL

I REPLACE 3. CARDS
WHAT CARDS DO YOU WANT REPLACED ...

    C6

YOUR NEW CARDS ARE     S6


I BET 4.
THE POT EQUALS 16.
YOUR BET ...

    12


I CALL
MY HAND IS     D14 H13 H11 S13 S7
YOU WIN
MY SCORE IS 10.
YOUR SCORE IS A MERE -10.
I WIN THE GAME

NIL
```