MIT/LCS/TM-20

A COMPUTER MODEL OF SIMPLE FORMS OF LEARNING

Thomas L. Jones

January 1971

TM-20

A COMPUTER MODEL OF SIMPLE FORMS OF LEARNING

Technical Memorandum 20

(This report was reproduced from a Ph.D. Thesis, MIT, Dept. of Electrical Engineering, September 1970.)

Thomas L. Jones

January 1971

PROJECT MAC

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Cambridge Massachusetts 02139

Acknowledgments

I would like to thank Professors Seymour Papert and Wayne Wickelgren for their assistance and encouragement in getting this project started; Professor Marvin Minsky for rescuing it from derailed states on several occasions; B. Smith for spotting a most serious bug in time to do something about it; J. Kohr for writing a version of one of the subroutines; and Marcia Murphy and Margaret Shaw for typing the manuscript.

This work was supported by the Artificial Intelligence Laboratory, an M.I.T. research program sponsored by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract number N00014-70-A-0362-0002.

Abstract

A basic unsolved problem in science is that of understanding learning, the process by which people and machines use their experience in a situation to guide future actions in similar situations. This thesis presents an approach to the learning problem and a learning-oriented approach to the artificial intelligence problem. These approaches are illustrated in a computer program called INSIM1, which models simple forms of learning analogous to the learning of a human infant during the first few weeks of his life, such as learning to suck the thumb and learning to perform elementary hand-eye coordination.

The program operates by discovering cause-effect relationships and arranging them in a goal tree. For example, if A causes B, and the program wants B, it will set up A as a subgoal, working backward along the chain of causation until it reaches a subgoal which can be reached directly; i.e., a muscle pull.

The work is discussed in relation to fundamental scientific issues, and proposals are made for future research.

Table of contents and chapter abstracts

This chapter describes the background of the artificial intelligence problem and presents an approach to solving it. Briefly, the idea is to incorporate into an artificial entity that which an infant's brain has in it which enables him to learn about the world and develop some degree of control over it, then allow the artificial being to go through similar phases of infancy, childhood, and adulthood.

Chapter 1: The Artificial Intelligence Problem--An Overview

The idea of an artificially intelligent being has a long history in literature, folklore, and science fiction. From the legend of the Golem through Karel Capek's "robots" to the recent movie, "2001," the human imagination has been gripped by the concept of a machine endowed with a mind, thoughts, and feelings. Fictional artificially intelligent beings usually exhibit an odd mixture of intelligent and very unintelligent aspects of behavior; their speech sounds like a pre-recorded announcement; their personality is cold and "machinelike;" and, of course, they often become hostile. One presumes that real intelligent machines will seem very "human" and not "machinelike" at all. With the invention of the electronic computer, artificial intelligence moved out of the realm of fiction, and, starting around 1955, a technology of heuristic programming has been developing, with the development of programs which prove theorems in logic (Newell and Simon, 1956), plane geometry (Gerlernter, 1958), and group theory (Norton, 1966); a program which plays an excellent, if not yet master-level game of chess (1967); programs which do symbolic integration (Slagle, 1961; Moses, 1967), and recognize geometric analogies (1964), all of which are difficult problems for humans to solve. There has been a parallel effort in the direction of what may be

called "lower behavior," including cybernetic models of early learning (Becker, 1970), and learning in neural nets (Rosenblatt, 1958; Rochester, 1956). In the classic statement of the artificial intelligence problem by Minsky (1961), the program is to solve a hard problem by searching through some space of solution attempts, aided by heuristics such as learning, pattern recognition, planning, and induction.

It seems to me that the way to achieve artificial intelligence is to build an "artificial infant"(compare Turing, 1950), which would "grow up" in much the same way as a human child. Its development would be divided roughly into three overlapping phases:

Phase I:

This is the engineering phase of developing the artificial infant and requires most of the hard work. The artificial infant is to contain something analogous to the capabilities of a human infant's brain, which enables him to learn about the world and develop some degree of control over it. The machine would have a body with sensory and motor equipment for interacting with the real world.

Phase II: Sensorimotor learning

The artificial infant plays and explores, learning to relate sensory data to motor actions, coding the basic subroutines associated with the concepts of objects, space, and time (Piaget, 1952); this phase is to be compared with the

learning of a human infant aged 0-18 months. Note that the program must discover these matters for itself, with only minimal assistance from outside (such as doing interesting things for it to imitate).

Phase III: The education phase

In this period, the machine is to be compared with a human child who has learned to talk. The objective is to transmit to it a cultural heritage of concepts, values, goals, and facts. The major medium of communication is to be English or another natural language.

The education of an intelligent machine would have many similarities with ordinary programming; one would communicate with it in a language, specify problems, correct bugs, etc. The difference is major in that the innate and previously learned software must bear most of the burden of deciding the precise algorithms to be employed in a particular situation; the teacher would specify the problem and perhaps give some helpful but ambiguous characterization of the methods to be employed. The distinction is the distinction between a calculus textbook and a computer program for doing calculus problems such as the ones by Slagle (1961), Moses (1967), and Charniak (1969). There is a large body of information built into the program which is not in the calculus book at all (Heuristics, pattern-recognition methods, and the like), information which a human student discovers for himself and which one may reasonably expect an intel-

ligent computer program to discover for itself, given the proper innate programming and intellectual background.

Phase IV:

This phase corresponds to an adult human and is the phase in which the intelligent machine can solve hard problems; if we have done our engineering properly, it should be much faster than a human and much more adept at such skills as memorization and mathematical calculation, making it a very clever entity indeed.

This is obviously a long-term project; however, there is no reason why it could not be done given enough time, effort, and cleverness. The key unknown parameter is the amount of innate code needed to get the machine to bootstrap itself into intelligence. If the amount of innate code is of the order of 10,000 to 50,000 words, the software problem is the easy part and the hard part is developing hardware with sufficient speed and memory capacity. If, as I suspect, the innate code must be of the order of 50,000 to 1,000,000 words, we have a decades-long research project. (If a billion words of innate code are needed, the project is probably not feasible.)

Chapter 2: Review of the Literature

A. Piaget: The Origins of Intelligence in Children (1952)

This book, by the noted Swiss child psychologist, deals with the mental development of infants from birth to around 18 months. Piaget presents very detailed observations of three of his own children, the flavor of which is best given by an example:

Observation 16- At (age 1 month, 1 day) Laurent is held by his nurse in an almost vertical position, shortly before the meal. He is very hungry and tries to nurse with his mouth open and continuous rotations of the head. His arms describe big rapid movements and constantly knock against his face. Twice, when his hand was laid on his right cheek, Laurent turned his head and tried to grasp his fingers with his mouth. The first time he failed and succeeded the second. But the movements of his arms are not co-ordinated with those of his head; the hand escapes while the mouth tries to maintain contact. Subsequently, however, he catches this thumb; his whole body is then immobilized, his right hand happens to grasp his left arm and his left hand presses against his mouth. Then a long pause ensues during which Laurent sucks his left thumb in the same way in which he nurses, with greed and passion (pantings, etc.).

Piaget calls this period of the child's life the "sensorimotor" period, since the child is concerned with developing sensory and motor capabilities such as seeing and picking up objects.

Piaget analyzes the child's behavior in terms of schemas which correspond roughly to subroutines in a computer program; thus the "sucking schema," the "grasping-sucking schema," etc. As the child develops, schemas are formed, co-ordinated (compare: a higher level program is coded which uses one subroutine to prepare for another to operate properly), differentiated (compare: a new subroutine is coded by modifying and adding to a previously existing one), and generalized (compare: extending a subroutine to new cases).

It is fascinating indeed to read Piaget's description of infants in language which "makes sense" to computer researchers, and Piaget's work is a rich source of ideas for things to get machine learning programs to do.

B. Pavlov

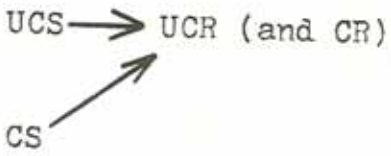
Pavlov (1927) developed one of the earliest and most famous theories of behavior learning. Pavlov regarded behavior as synthesized out of reflexes (stimulus-response connections). Pavlov's most famous experiment was performed on dogs. A dog's salivary glands will emit saliva when the dog sees or smells food. Pavlov's experiment consisted of repeatedly ringing a bell, then feeding the dog. After many repetitions, the bell became a "dinner bell;" i.e., the dog would salivate upon hearing the bell, in preparation for being fed.

Pavlov postulated that this behavior pattern was implemented as follows: Let the UCS (unconditioned stimulus, the sight and smell of food in this case) be connected to the UCR (unconditioned response, salivating in this case).

UCS $\longrightarrow$ UCR

Now suppose another stimulus S (the bell) occurs repeatedly before the UCS. Pavlov postulated that S would

become a CS (conditioned stimulus) connected to the response:



Pavlov regarded the reflex as the basic unit from which behavior is synthesized; thus the reflex plays a role in his system similar to the goal-subgoal link in INSIM1. Pavlov postulated a second signal system which he regarded as responsible for complex, voluntary behavior. Thus the "reflex of freedom;" also, he considered thinking to be a set of conditioned reflexes. Pavlov described a long list of characteristics of reflexes. If the unconditioned stimulus (the bell) occurs without the conditioned stimulus (the food), the conditioned response is gradually inhibited. The reflex is not forgotten, and it is restored after a delay of a few hours. This phenomenon is called spontaneous recovery.

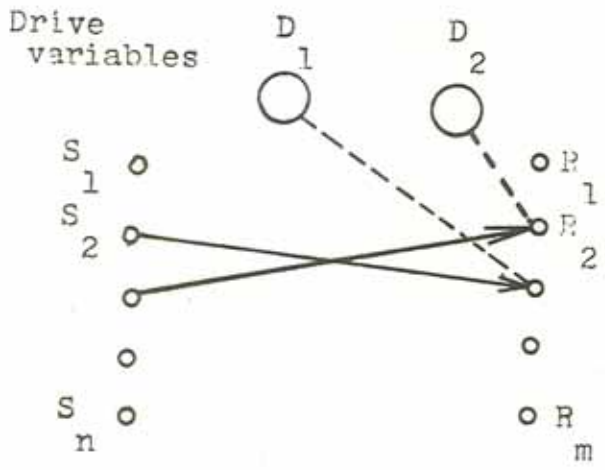
After the animal has been conditioned to respond to one stimulus, other, similar stimuli will elicit the same response. This process is called generalization. If the new stimulus is not reinforced, say, by food, a discrimination

process will occur, and the animal will learn to respond only to the original stimulus. Pavlov postulated a cortical process called irradiation which was to underlie generalization, and one called concentration to underlie discrimination.

To me, the basic problem with Pavlovian neuropsychology is its inability to handle adequately matters of motivation and goal-subgoal relationships.

C. Hull's "A Behavior System" (1952)

Hull's behavior model is one of the most interesting concepts originating from the behaviorist school of American psychology, the most famous member of which is Skinner (1953). The model attempts, with some success, to give a mathematically precise theory of simple forms of learning, such as a rat learning to press a bar to obtain food. There are seventeen geometry-like postulates which describe a network device, with stimulus (input), response (quasi-output) and drive (e.g., hunger) variables.



The idea is that a sensory input, such as seeing a light flash, activates an appropriate response, such as pressing a lever, provided the appropriate drive (hunger) exists. Whether or not a response is emitted depends on its reaction potential, a continuous-valued function of time. If the reaction potential of a response is greater than some threshold, and if it is greater than that of all incompatible responses (e.g., turn left vs. turn right), then the response is emitted. The reaction potential is the product of drive strength and habit strength ${}_{S}H^{*}_{R}$. The latter variable contains the learned information; ${}_{S}H_{R}$ is incremented whenever S and R bring reward and decremented

whenever S and R fail to bring reward. This, if one flashes a light and then feeds the rat after he pushes the lever, S^{H}_{R} will increase and the rat will be more likely to make the correct action the next time the light flashes.

The basic reason for using numerically valued reaction potentials is that it handles the goal-conflict problem in a very direct and plausible way: If two goals conflict (e.g., desire for food and fear of an electric shock), the one which is "strongest" wins. (I claim that this numerical weighing of values belongs at a different interpretive level entirely.)

Hull's theory was very precisely worked out, and there is no obvious reason why it does not constitute an algorithm. If it does, it is the only such learning theory except for INSIM1. It is surprising that the psychologists have not attempted a computer implementation of Hull's theory in order to see if it is really an algorithm, and if so, what its performance "bugs" are. I suspect that the treatment of subgoals is unsound in that Hull did not provide a separate variable for the subgoal; instead, the subgoal is a partial activation of a response variable. Also, Hull did not handle stimulus interaction or concept formation.

If nothing else, INSIM1 has demonstrated the feasibility of computer models of learning theories. I hope that psychologists will adopt this method of testing their theories, as opposed to rhetoric, persuasion, and manual methods, which are much more clumsy and much less likely to reveal performance "bugs."

D. Other behaviorist learning models

In addition to Hull's behavior system, there are several other interesting learning theories of the behaviorist school. (See Hilgard (1956) for an excellent survey of learning theories.) Thorndike (1898) antedated the behaviorists and provided several concepts which have greatly influenced learning theorists. Thorndike's most famous experiment consisted of confining a hungry cat in a box with food outside. The cat must operate a latch to escape. On the first few trials, the cat dashes around, claws, and bites in a manner which some observers have characterized as "random." (Whether or not there is really a randomization process involved is unknown.) On later trials, the cat learns gradually to operate the latch with less and less

delay. The gradualness of this learning suggests the familiar statistical coefficient learning.

Thorndike explained behavior as the result of "bonds" or "connections" between sense data and actions and thus might be considered to be an originator of stimulus-response psychology. S-R psychology has been criticized, I think correctly, (e.g., by Miller, Galanter, and Pribram (1960)) as failing to adequately take into account the purposiveness of behavior. However, its influence has been large and often turns up in unexpected places. (The PSIM subsystem of INSIM1 was motivated to a large degree by a desire to devise a computer language which was "stimulus-response" oriented.)

Thorndike proposed (and later abandoned) three laws governing the formation of S-R bonds: the law of effect, which stated that behavior which is followed by reward is likely to be repeated; the law of exercise, which took into account the amount of practice*; and the law of readiness, which included the functions which we would today assign to a goal-subgoal system.

---

*The later, simple formal mathematical models of Bush and Mosteller (1955) and Estes (1967) attempt to "explain" the need for practice; Bush and Mosteller show how, in these models, the gradual coefficient-learning theory is actually isomorphic (mathematically indistinguishable) with the all-or-none "stimulus-sampling" connection theory of Estes.

Watson is often considered to be the founder of the behaviorist movement in psychology, which takes overt behavior as its subject matter, rather than conscious experience. Behaviorism arose as a revolt against the rather sterile tradition of introspective psychology prevalent around the turn of the century. From a modern viewpoint it is probably good strategy in learning theory to study simple forms of behavior and to neglect conscious experience, since behavior is easier to handle in a learning model. But behaviorism must be regarded as methodologically inadequate because of its rejection of introspective reports, which can be used to great advantage; see, for instance, Newell and Simon (1958).

Skinner (1953) has advanced one of the most famous of the recent behaviorist learning theories. In Skinner's system, the fundamental unit of behavior is called an operant. Examples are eating a meal, writing a letter, and driving a car. In Skinner's model, operants are emitted according to a stochastic process, and the key parameter is

the probability of emitting a particular operant under various stimulus conditions. Certain stimuli, such as food and water, are called reinforcing stimuli. The fundamental law of learning in Skinner's system is that if the occurrence of an operant is followed by presentation of a reinforcing stimulus, the strength (probability of emission) of the operant is increased. Imagine a rat in a box where pellets of food will be given it if it presses a bar with its foot. At first, the rat presses the bar by chance and is fed (reinforced). Then the probability that it will press the bar (emit the operant) is increased.

The behaviorist learning theories have had great practical value in providing the sort of semi-empirical understanding needed in education. It is no accident that several clever innovations in educational technology, such as Skinner's teaching machine approach (1968), have been made by behaviorists. From a theoretical viewpoint, however, the behaviorist theories, except possibly for Hull's, are subject to the usual criticism made by computer scientists about psychological theories: That they are on a different level of abstraction from what is needed for a precise, formal model, and that they are too vague to be made the basis of an algorithm. Dozens of attempts have been made to develop computer models of learning, based on behavior

theory (I myself have made a dozen or so), without noticeable success. One can only conclude that, except for Hull's, these behaviorists models are not suitable bases for formal theories.

E. Hebb: Organization of Behavior (1949)

In contrast with Hull's model, where the influence of neurophysiological data was just below the surface, Hebb constructed a learning theory which makes explicit reference to neurons which fire, sending electrical impulses down nerve fibers to synapses through which the information is transmitted to other neurons. Hebb made a basic postulate which may be understood through the example of an experiment in which the subject hears a list of pairs of nonsense syllables; the goal is to remember the second syllable, in response to the first, or cue syllable. Assume that we have cells which fire, corresponding to various properties of the syllables (Wickelgren, 1966). Hebb postulated a "fire together, stay together" rule; if cells A and B fire at the same time (hearing the pair KNC-JLZ), the connection joining them will be strengthened so that if A fires again (hearing the cue KNC), B will fire also (retrieving JLZ).

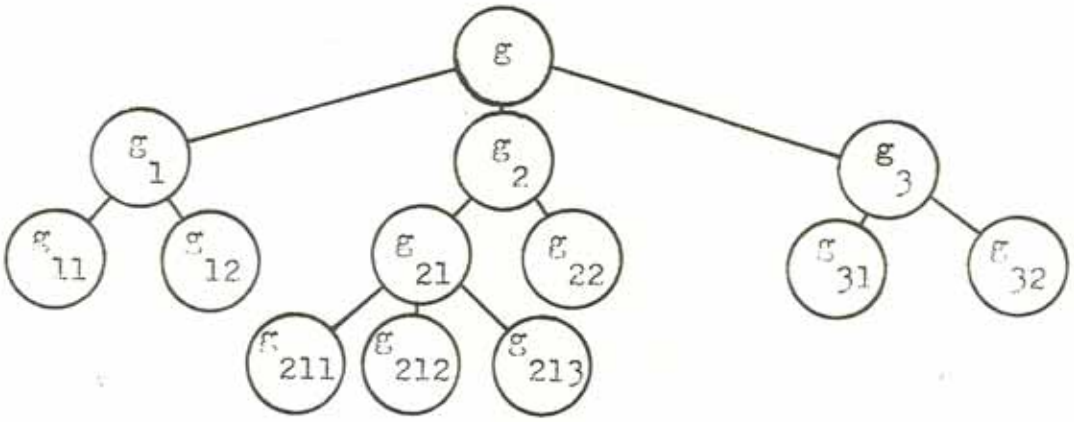
Hebb suggested that cells with a fire-together stay-together rule would combine to form hierarchical cell assemblies, the neurological analog of subroutines in a computer program. He gave a coherent account of how such cell assemblies could develop the ability to recognize a visual pattern such as a triangle.

A computer simulation by Rochester, et al (1956) showed the incompleteness of Hebb's model. The work of Hebb, McCulloch, and others stimulated a great deal of interest in "self-organizing neural nets" (Farley and Clark, 1954; Shimbel; Ashby, 1952).

F. MULTIPLE

MULTIPLE (MULTI-Purpose theorem prover that Learns) (Slagle and Bursky, 1968) is a program which solves problems (such as end-games in kalah) by proving theorems. The program starts with a main proposition (compare: main goal) and generates sub-propositions which, if proved, would allow the main proposition to be proved. The propositions may be

displayed in a tree (compare: goal tree).



The process of generating subpropositions is called sprouting. The basic decision which MULTIPLE must make is that of which proposition to sprout from. MULTIPLE does this by assigning a numerically valued merit to each untried proposition. The concept is that a proposition has merit to the extent that proving it is likely to change the probability of the top proposition and if it is inexpensive to prove. More precisely, the program selects the proposition which maximizes $\Delta p/c$, where Δp is the (estimated) change in the probability of the top proposition and c is the estimated cost of sprouting. The numerical heuristics yield an efficient, but not theoretically optimal proof search.

G. Winston's program (1970)

This program is capable of learning simple visual categories, such as an arch, from examples. The program has a descriptive language which is the output of an analysis of the structure of an object.

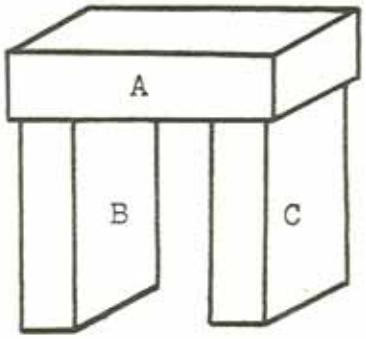


Figure 1

Here is how Winston's program would learn the concept of an arch: First, we give it the object in Figure 1 and tell it that the object is an arch. Next, we give it the object of Figure 2 and tell it that this object is not an arch.

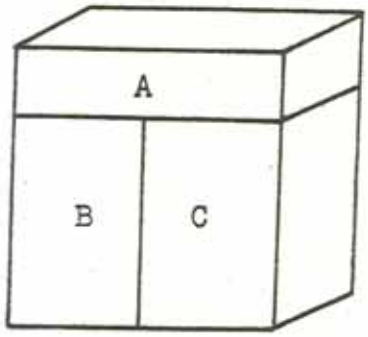


Figure 2

The program AND's into its list of requirements for an arch the rule that the two columns must not abut one another.

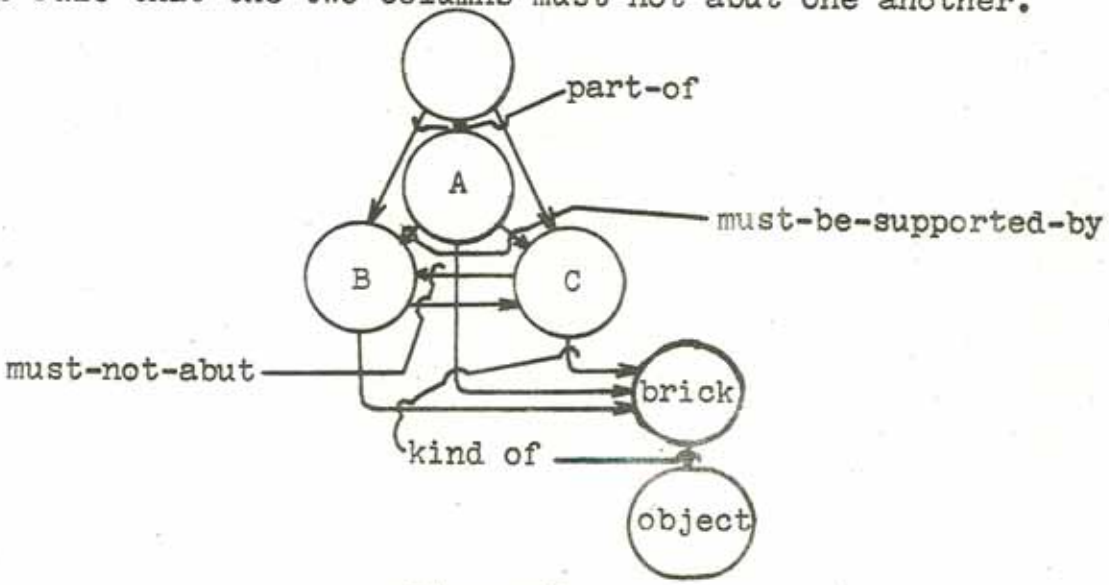


Figure 3

This requirement is put in because the difference in position is the "most prominent" difference between the objects of

Figure 1 and Figure 2. Next, we present it with the objects of Figures 4 and 5. The program learns that an arch must have the top-object supported by the columns, and that the top-object need not be a brick.

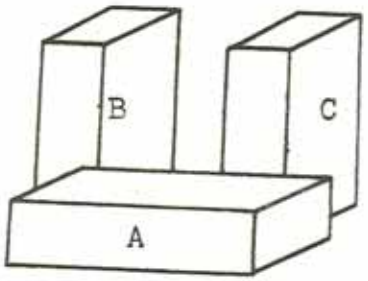


Figure 4

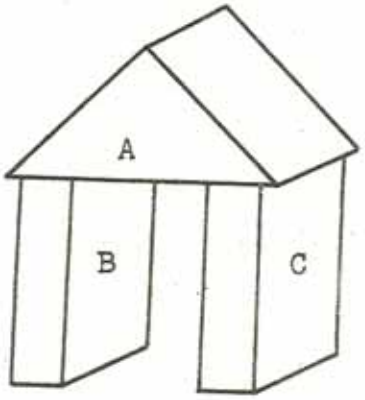
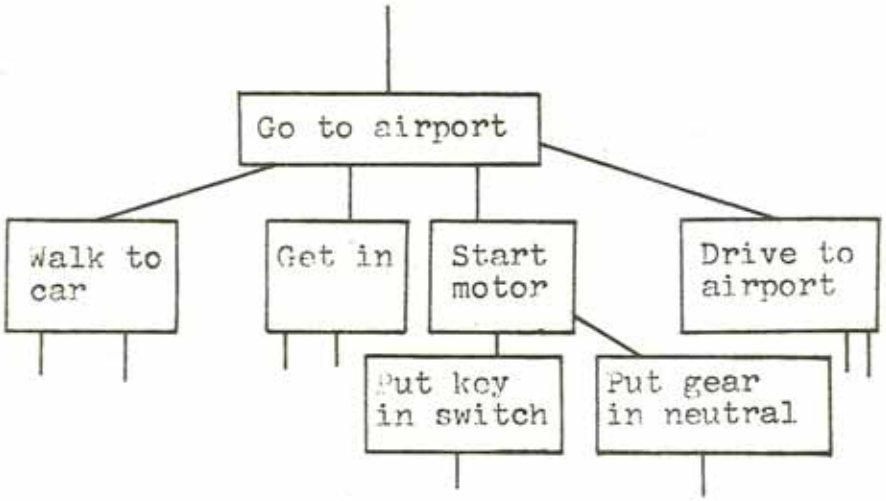


Figure 5

Winston's program includes many arbitrary elements in assigning priorities to differences and in matching of descriptions, and one has trouble understanding just what the program does. Nevertheless, I believe that Winston's work illustrates that there is a quantum jump in learning ability which becomes available when a program can decompose the universe into entities.

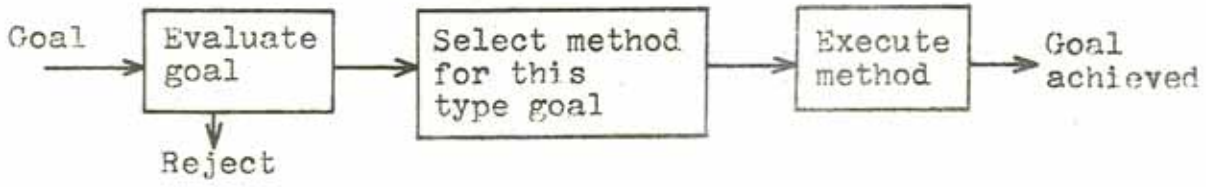
H. GPS

The Generalized Problem Solver of Newell, Shaw, and Simon (1959) is an organizational framework within which a variety of problem-solving processes may be mechanized. GPS is a teleological (goal-seeking) device in which a difficult goal is achieved by reducing it to simpler subgoals, each of which may be subdivided. This process is a form of planning which is familiar in common-sense thinking. "I want to go to the airport. Therefore, I will walk to the car, get in, start the motor, and drive it to the airport. To walk to the car, I will---"



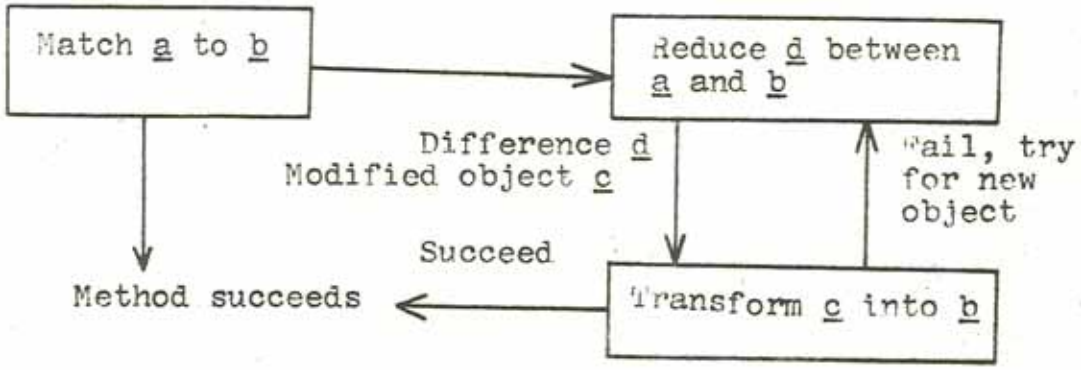
In the GPS formalism, a problem is described in terms of objects and operators which transform one object into another. For a problem involving actions in the physical world, the objects would be state descriptions and the operators would be actions. If the problem is to prove a theorem in mathematical logic, the objects would be expressions, and the operators would be rules of inference.

The executive organization is simple.



Of course, the hard problem in such a system is to find subgoals which will aid in achieving the goal at the next higher level. Goals are classified into three types, with a corresponding method for each type. The problem is usually posed as a goal of type 1: Transform object a into object b.

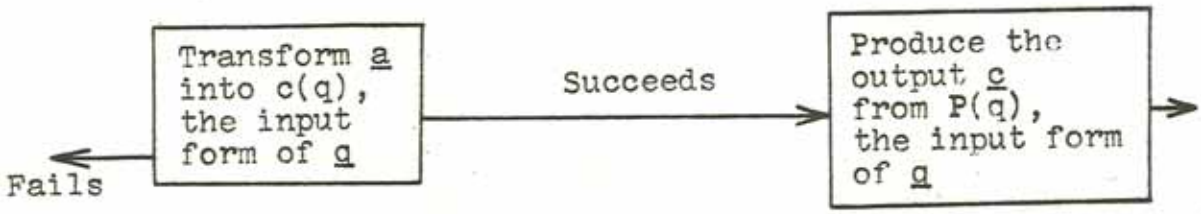
Goal type No. 1: Transform object a into object b



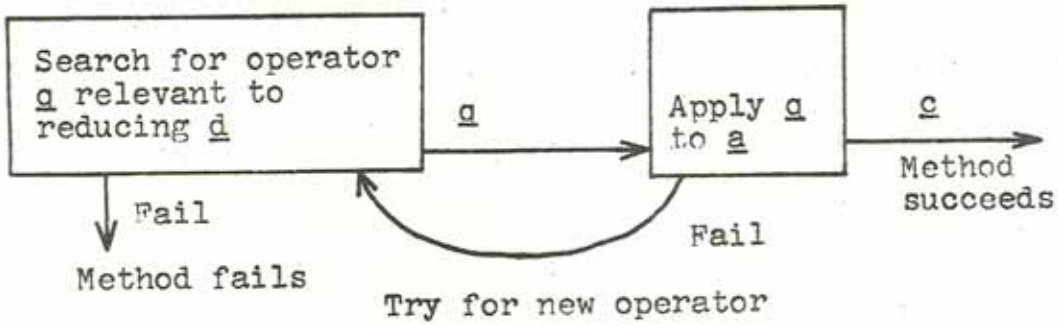
The two objects being tested are matched and a pattern recognition process is used to find several differences

between them. For example, if the problem is to transform one expression in symbolic logic into another, the differences might be that the two expressions had different main connectives or that the terms were grouped differently. A table of difference priorities is used, causing the machine to attack the hardest differences first. The process of reducing differences involves two other goal types.

Goal type no. 2: Apply operator q to object a



Goal type no. 3: Reduce the difference d between objects a and b



A table of differences vs. operators is used to tell the machine which operators will help in reducing a given difference. The process of reducing differences imparts a "directionality" to the program, allowing it to progress toward the goal in a sequence of steps.

In addition to proving theorems in symbolic logic, GPS has been used successfully to prove trigonometric identities and solve simple puzzles.

J. Newell, Shaw, and Simon: A Variety of Intelligent Learning in a General Problem Solver

The Newell, Shaw, and Simon paper, "A Variety of Intelligent Learning in a General Problem Solver" (1959) is an enigma among scientific papers. It presents an intricate proposal for getting GPS to solve adult-level problems, such as proofs in symbolic logic, on a heavily self-organizing basis, with little manually prepared problem-specific information. The enigma lies in the fact that the 1959 proposal has not, as of 1970, been implemented, and that, if implemented, it would constitute a significant advance

over any learning code available in 1970, including INSIM1.

The "Intelligent Learning" paper proposes a simple method for learning the operator-difference table and a much more complicated method for learning the differences themselves. The method for learning the operator-difference table relies on the fact that operators are stated in what may be called a "theorem" form, such as $A \vee B \rightarrow B \vee A$ (operator R1 in the logic task environment). To find out that this operator is relevant to a difference in <u>order</u> of the subexpressions, one need only apply the difference computing subroutine to the left and right sides of the expression which describes the operator and note that $A \vee B$ differs in order from $B \vee A$. Note that no use has been made of experience on actual problems.* By contrast, INSIM1 must laboriously test its operators (performance subroutines) in actual situations to see what difference they make in the environment.

---

*But if the operator had been described by almost any kind of procedural form, like "If CAR E equals OR then swap CADR E and CADDR E," testing on examples would be needed.

A much harder and more interesting problem is to get GPS to invent a good set of differences. In order to do this, the authors invented a language called DPL (difference processing language) in which difference subroutines are to be written. The elements of the language are:

PROCESSES FOR SYMBOLIC LOGIC ENVIRONMENT

Symbol	Name	Input	Output	Examples
t	Find terms	object	set of all sub-objects	t(P∨Q)=(P∨Q), P⊃Q
l	Find left	object	left-hand sub-object (∅ if doesn't exist)	l(P∨Q)=P l(Q)=∅
r	Find right	object	right-hand sub-object (∅ if doesn't exist)	r(P∨-Q)=-Q
c	Find connective	object	main connective of object (∅ if object is variable)	c((P⊃Q)∨R)=∨
s	Find sign	object	main sign of object	s(-(R∨P))=- s(-R∨P)=+
v	Find variable	object	variable letter (∅ if a compound object)	v(P⊃Q)=∅ v(-Q)=Q
b	Test if constant	anything	input, if constant (∅ if it contains free variables)	b(P∨Q)=P∨Q b(A)=∅
f	Test if free variable		input, if a free variable (∅ if not)	f(A)=A f(P)=∅
⊃	Test if ⊃	anything connective	⊃ if connective is ⊃ (∅ if not)	⊃(⊃)=⊃ ⊃(∨)=∅

PROCESSES FOR SYMBOLIC LOGIC ENVIRONMENT
Cont'd

Symbol	Name	Input	Output	Examples
v	Test if v	connective	v if connective is v (ø if not)	v(v) = v v(>) = ø
.	Test if .	connective	. if connective is . (ø if not)	.(.) = . .(v) = ø
+	Test if +	sign	+ if sign is + (ø if not)	+(+) = + +(-) = ø
−	Test if -	sign	- if sign is - (ø if not)	−(−) = − −(+) = ø
A(B,C,..)	Test if A (B,C,..)	variable	A if letter is A (ø if not)	A(A) = A A(P) = ø
P(Q,R,..)	Test if P (Q, R,..)	variable	P if letter is P (ø if not)	P(P) = P P(Q) = ø

GENERAL DPL PROCESSES

Symbol	Name	Input	Output	Examples
A [X]	Assign	any object	X if input is not ø, ø if input is ø	A[+](P,Q) = + A[+]ø = ø
$\bar{A}$ [X]	Inverse assign	any object	X if input is ø, ø if input not ø	$\bar{A}$[+](P,Q) = ø $\bar{A}$[+]ø = +
B [X]	Blank	any object	goes through all subparts of input: x. If X (x) not ø, replace X(x) by ø in input	B[c](P⊃(Q∧R)) = Pø(QøR)
C	Find component	set	takes any component for output	C{⊃, v, .} = v

GENERAL DPL PROCESSES
Cont'd

Symbol	Name	Input	Output	Examples
D	Difference	any pair of objects	compares corresponding subparts of the two inputs. If equal, replaces each by Ø. Output is modified pair.	D((P,Q,R),(Q,Q,R,P)) =((P,Ø,Ø),(Q,Ø,Ø,P))
E	Expand	set of sets	output is set of all elements in the subsets of input, with multiplicity	E{{P,Q};{P,R}} =(P,Q,P,R)
F	Find first	list	first item on the list (Ø if doesn't exist)	F(P,Q,R)=P
G[X]	Group	set	output is a set of sets. Each subset contains all the items of the input set with the same value of X(x)	G[I](P,P.Q,P,R,Q)= ((P,P,P),(Q,Q (R))
I	Identity	any object	input	IX=X
K[X]	Constant	any object	X, for any input	K[+]P=+
L	Find last	list	last item on the list	L(P,Q,R)=R
M	Find prior	item from list	item preceding input item (Ø if doesn't exist)	MP from (Q,R,P,S)=R

GENERAL DPL PROCESSES
Cont'd

Symbol	Name	Input	Output	Examples
N	Find next	item from list	item following input item (ø if doesn't exist)	NP from (Q,R,P,S)=S
P	Intersection	set of sets	set of items common to all subsets of input	P{{P,Q}, {Q,R}}={Q}
R[X]	Select representative	set	set consisting of one representative element of each value of X(x). Compare G[X]	R[I]{P,P,Q, P,R,Q}= (P,R,Q)
S[X]	Select	set	set of items of input set with X(x) not ø	S[v]{P.Q, -R, R=P, R} ={-R,R}
U	Set	list	set of items on list	U(P,Q,R)= {P,Q,R}

For example, the subroutine D(R[I]vt)* detects a difference between the lists of variables which occur in two expressions.

A key concept in the "Intelligent Learning" paper is that GPS is used recursively to solve the human-level problem of learning (i.e., discovering) a good set of differences. There is a separate task environment called

the B-environment where the objects are sets of differences. The B-operators are: (note: an A-difference is a difference pertaining to the performance environment, logic theorems in this case)

B-Operators

Q1 Add an A-difference that gives + for pair X and ∅ for pair Y. (A pair may either be a pair of objects, or the condition and product forms of an operator.)

Q2 Modify A-difference T to give + for pair X.

Q3 Modify A-difference T to give ∅ for pair X.

Q4 Delete A-difference T from set S.

Q5 Add an A-difference that gives + for pair X.

Apparently the B-operators are intended to be of the character of goals rather than tidy little subroutines. There is a set of B-differences which express the characteristics of a good set of A-differences:

B-Differences

D1 The set of A-differences not consistently defined for some pair of objects.

D2 The set of A-operators with no associated difference.

D3 The set of A-object pairs with no associated difference.

B-Differences (cont'd)

D4 The set of non-orthogonal situations (each situation consists of an A-object, a list of A-operators, the product from applying the operators to the given A-object, and the new differences between the input and output that are not associated with any of the operators).

D5 The set of full A-differences (having all A-operators associated with them).

D6 The set of empty A-differences (having no A-operators associated with them).

D7 The set of A-differences with more than one associated A-operator.

D8 The set of A-operators with more than one associated A-difference.

D9 The total number of A-differences.

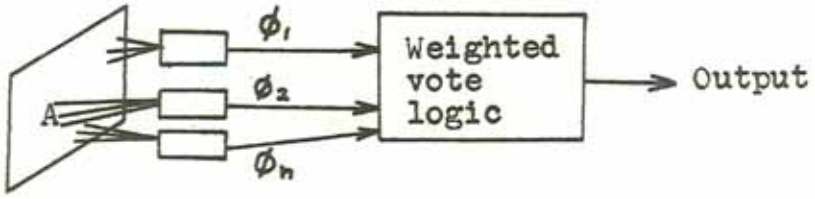
The paper concludes with a rather sketchy hand simulation of how the program might learn a set of differences for the logic environment.

It is impossible to evaluate the soundness of the proposal without actually trying to program it and see where the gaps are. It is quite unclear what is needed to make it work or what degree of success can be achieved. The paper underscores the importance of the "dirty work" of artificial intelligence research. Without programming, debugging, and experimenting on the program, the project remains a question mark, although "on paper" I consider this to be the most exciting and promising of the GPS studies.

The next two sections discuss several computer-oriented learning techniques which are dissimilar to each other in many ways, but are lumped together because they have the same general formal character and could be interfaced into a learning code of the INSIM type in similar ways. Each of these programs attacks in a different way what may be called the signal prediction problem in learning codes. Formally, they emit predictions about some variable, e.g., a variable which has the value 1 if an object is in a certain category (such as a pattern which is an instance of the letter A), the value 0 otherwise). The predictions are made on the basis of some set of basis variables $b_1, b_2, \ldots, b_n$, such as a set of visual properties related to the object (e.g., whether or not it has a vertical stroke).

K. The Perceptron

The term "perceptron" (Rosenblatt, 1958) refers to a class of pattern recognition devices using a simple linear weighted-vote technique. A typical perceptron consists of a retina on which a pattern is projected, a set of feature recognizers which compute the predicates ϕ_i of the retinal pattern, and a weighted vote mechanism which outputs 1 if, and only if, $\sum \alpha_i \phi_i(X) > \theta$ where θ is the threshold, the α's are weight factors, and X is the pattern.



For example, suppose the perceptron is looking for the letter A. $\alpha_i \phi_i$ is, roughly, the weight of the evidence provided by the feature ϕ_i about whether or not the pattern is an instance of the letter A. If the total evidence exceeds the threshold θ, the perceptron says that the pattern is an A.

Perceptrons can be equipped to learn the coefficients α_i in much the same manner as discussed in Appendix 1 on coefficient learning. As pattern recognizers, they work well or poorly, depending largely on how well the features ϕ_i match the distinguishing properties of the pattern. Despite its name, the perceptron is only a small beginning of the solution of the pattern recognition problem; either the real perception capability must "live" in the functions ϕ_i, or else they must be put together in some much more versatile way than as a mere linear threshold function.

The amount of research and literature on the perceptron is truly breath-taking. Numerous papers and two books,

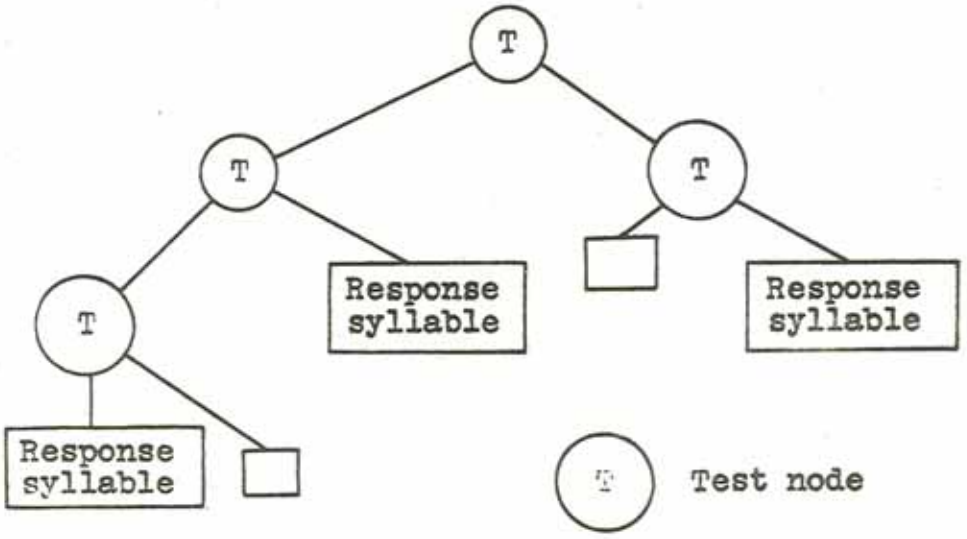
by Rosenblatt (1962) and Minsky and Papert (1969), have been written, bristling with equations, theorems, and proofs about what various types of perceptrons can and cannot do. Because of its simplicity, the perceptron lends itself well to mathematical analyses. The perceptron convergence theorem gives an algorithm for finding the coefficients α_i which is guaranteed to find the set of coefficients if a set exists which will identify the pattern in question (such as the letter A) from the set of functions $\phi_i(X)$, although it will not necessarily find it in a reasonable length of time. Minsky and Papert prove several theorems about what certain types of perceptrons cannot learn to do. We have little mathematical theory for systems like INSIM1. The evidence of its abilities is experimental, and I do not have proofs of its inability to learn to solve any types of problems. (It is clear to me that INSIM1 cannot learn to do, say, vision problems simply because it lacks the necessary machinery; it cannot parse rows of points into lines or arrays of lines into scenes.)

L. EPAM

Feigenbaum's EPAM (Electronic Perceiver and Memorizer) (1963) program is a model of simple forms of one-trial learning; i.e., learning which occurs after just one

experience rather than by a slow, statistical process as in the INSIM1 coefficient learner.

Feigenbaum modeled a verbal learning experiment of the type described previously, in which the subject is asked to remember associated pairs of nonsense syllables and recall the second syllable upon presentation of the first as a cue. EPAM grows a discrimination tree of tests which are applied to the due syllable; retrieved syllables are stored at the bottom nodes of the tree.



An example of a test would be "Is the first letter D?" Feigenbaum developed a very efficient algorithm for building these trees and retrieving information from them. The behavior of EPAM in learning (and forgetting) nonsense syllable pairs was compared with that of human subjects and found to agree quite well.

Adding a modified version of EPAM into an advanced program of the INSIM type would greatly improve its capabilities because of the ability of EPAM to learn in one trial. To see how this might possibly be done, define the cue basis of an EPAM system to be the ordered list of tests (noticing order) (T_1, T_2,...T_n) which EPAM inserts into the tree. EPAM systems perform well if the cue basis corresponds to the properties of the entities with which the system is dealing. The idea that there is such a thing as an entity (such as a syllable) whose properties "belong together" is so commonplace that its importance is easily overlooked. For example, in a real psychological experiment, the subject is bombarded with stimuli which have nothing to do with the experiment. If EPAM were to include, in its decision tree, tests about stimuli relating to the states of the experimenter's eyebrows or whether or not there were birds visible in the laboratory window, the performance of the EPAM system would be greatly degraded. Feigenbaum's program had a great deal of specialized knowledge about entities coded into it innately. A system of the INSIM type would need to have software to learn this entification for itself. Once the cue basis was learned, the EPAM tree could learn in one trial that some set of entity properties, such as blue and round,

gave a high Pr (B| A) for some link, such as shake→rattling sound, if the object were a rattle.

While I believe it feasible to implement an entification learner, it would be premature to speculate on how this system would work.

Chapter 3: INSIM1:
A Computer Model of Simple Forms of Learning

A. Introduction and Summary

INSIM1 is a computer program, written in LISP (McCarthy, 1960) on the ITS time-sharing system, which models simple forms of learning analogous to the learning of a human infant during the first few weeks of his life, such as learning to suck the thumb and learning to perform elementary hand-eye coordination.

The program operates by discovering cause-effect relationships and arranging them in a goal tree. For example, if A causes B, and the program wants B, it will set up A as a subgoal, working backward along the chain of causation until it reaches a subgoal which can be reached directly, i.e., a muscle pull.

A typical problem is the one-dimensional, three-point thumb-sucking problem, which can be described in logical notation as follows:

(1) object touching mouth $\rightarrow$ pleasure

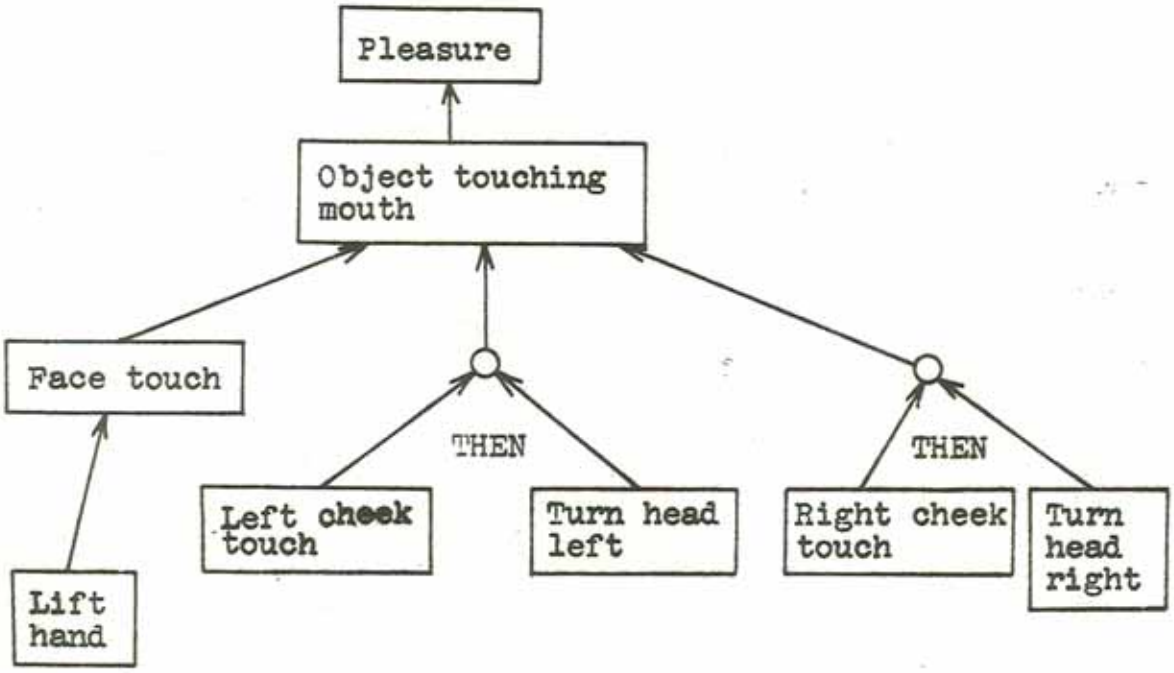
(2) (left check touch $\wedge$ turn head left) $\rightarrow$ mouth touch

(3) (right cheek touch $\wedge$ turn head right) $\rightarrow$ mouth touch

(4) (left cheek touch $\vee$ right cheek touch) $=$ face touch

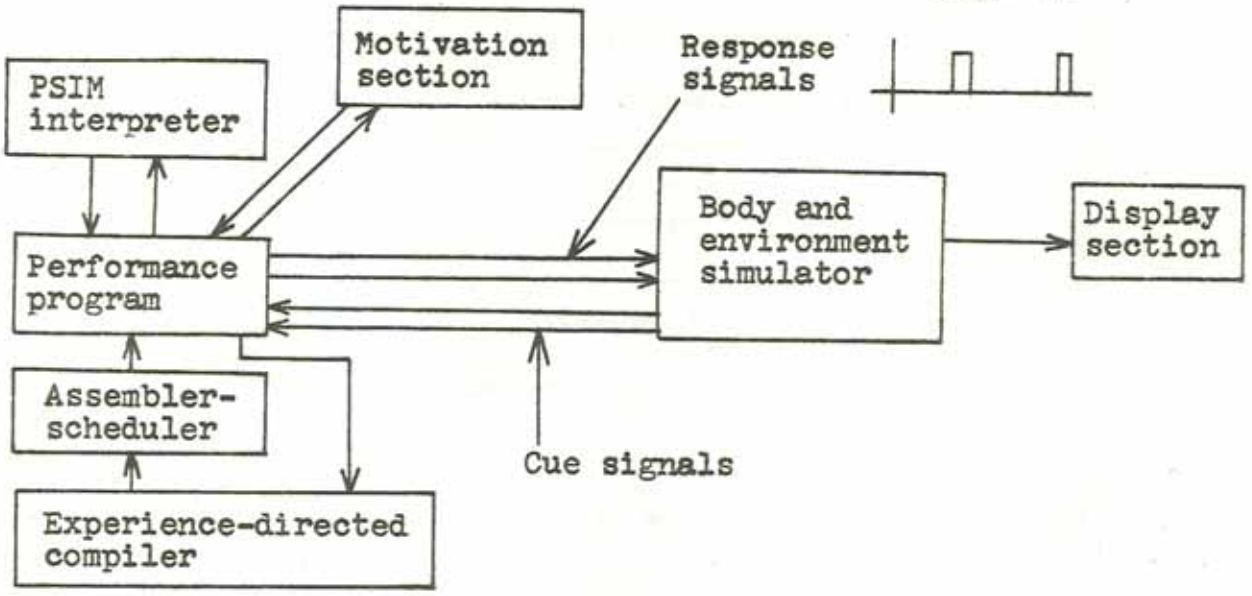
(5) face touch→mouth touch (sometimes)

(6) lift hand→face touch



After the program has learned these connections, it will emit the behavior sequence "lift hand, turn head (left or right)," resulting in pleasure.

Below is a block diagram of INSIM1:



The performance program has the direct responsibility for synthesizing behavior. It is written in an interpretive language called PSIM (parallel simulator). The performance program receives stimuli from and sends responses to a body and environment simulator; the display section provides real-time monitoring on the cathode-ray tube. The motivation section activates the main goal (oral gratification or curiosity).

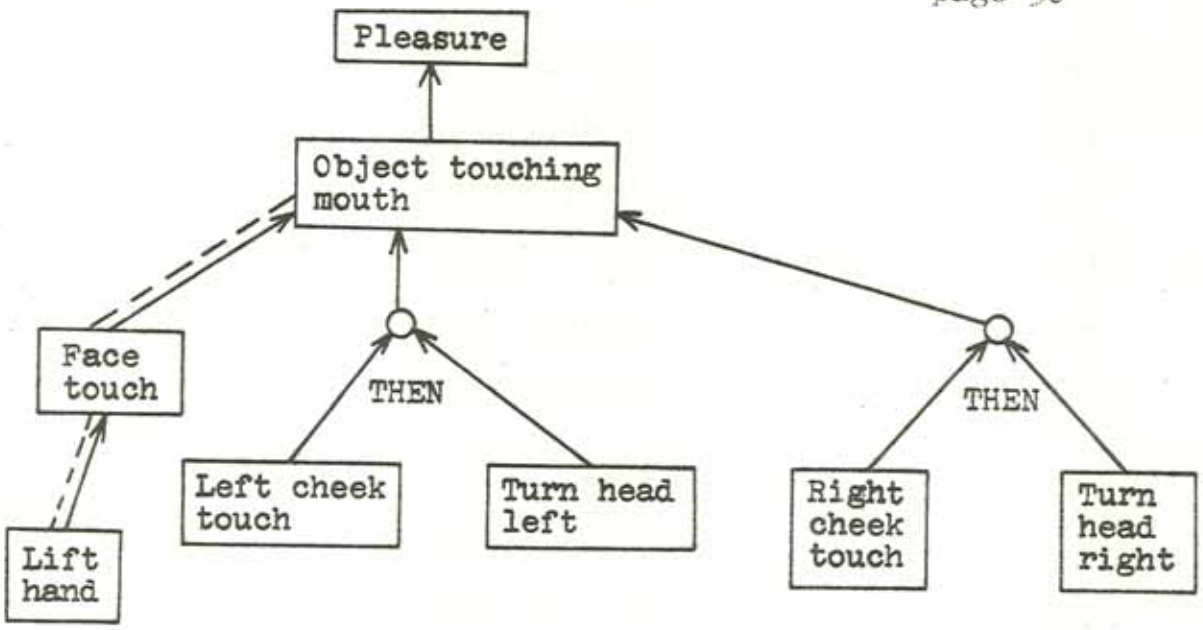
Relatively little of the performance program is innate. Most of it is generated by an experience-driven compiler, which is the core of the learning part of the program.

Causality is detected by statistical correlation; if a signal occurs on line A followed by one on line B, and if this sequence is repeated sufficiently many times, the

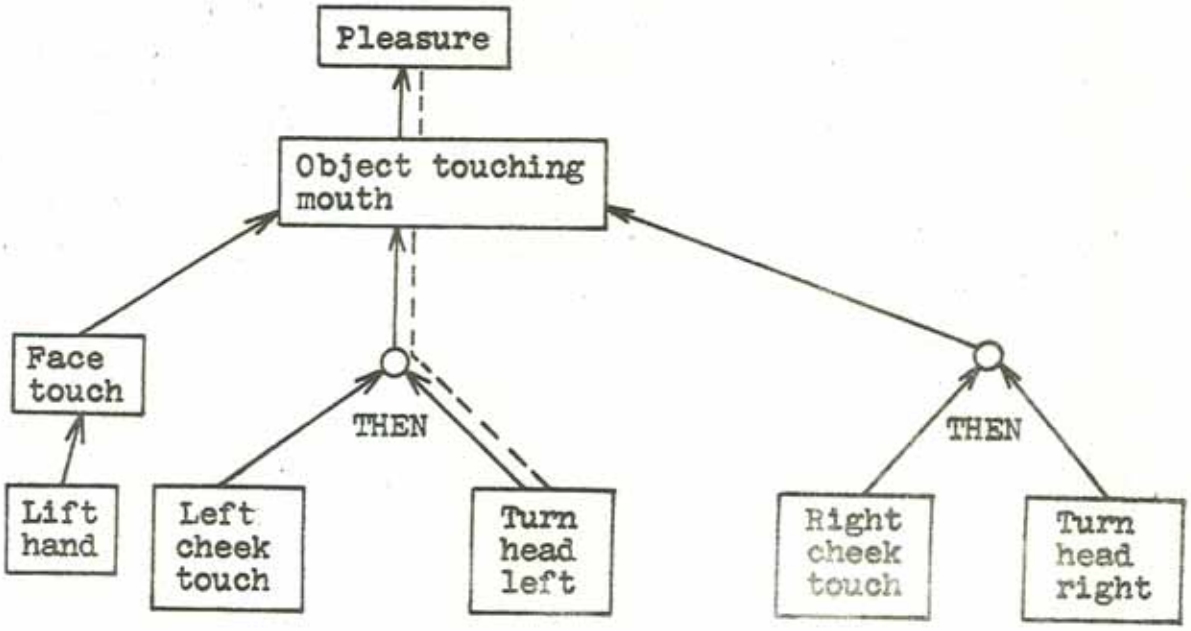
program assumes that A causes B. The program is equipped for the simplest type of pattern recognition and concept formation: the formation of logical AND's and OR's of previously known variables. The program has an intellectual motivation system which causes it to exhibit simple forms of curiosity, play, and exploratory behavior.

B. The Performance Program

As described above, the performance program has the direct responsibility for receiving cues from the environment and emitting properly timed and sequenced behavior. It is coded in PSIM, a language which will be described in detail below. The performance program operates by activating various branches of the goal tree at the appropriate times. In the thumb-sucking problem, assume that the motivation section has activated the main goal "oral gratification." The first step is to activate the extreme left branch of the tree (the dotted line indicates activation):



The "lifthand" response at the bottom of this branch is emitted to the body and environment simulator. After a delay of roughly two simulated-time seconds, a cue, e.g. "left cheek touch," comes back, indicating that the simulated hand has been lifted to touch the (simulated) left cheek. Next, the branch ending in "turn head left" is activated:



A "mouth touch" signal comes back from the body and environment simulator, indicating that this goal has been reached; the motivation section activates the oral gratification flag, "rewarding" the program for its successful effort.

The basic problem is to decide which branch of the goal tree to activate. (INSIM1 performance programs allow only one branch to be active at a time; hence there is no way to work on two goals simultaneously.) In a given situation, the decision is made in two phases, a feasibility study phase and a choice phase.

In the feasibility study phase, each branch of the tree is assessed, and an estimate is made of which branch is the quickest and surest way to the main goal. Two numerical quantities are computed for each subgoal, GPR (global success probability) and a GC (global cost). The GPR of a subgoal is an estimate of the conditional probability that, if the program attempts to achieve the subgoal, it will succeed in reaching it.

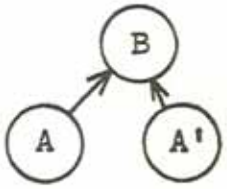
1. Computation of GPR and GC

This section is devoted to a detailed discussion of how GPR and GC are computed. On a first reading, readers may skip to Section 2 on the choice phase.

GPR is defined recursively as follows:

(1) For a "response" (directly controllable) variable, such as "lift hand" or "turn head left", GPR=1.

(2) Suppose that A is one of several OR'ed subgoals of B:



If A is the "best" subgoal according to a criterion to be presented momentarily, then GPR(B) = GPR (A) Pr (B | A), where Pr (B | A) is the conditional probability of B given A (i.e., the probability of getting from A to B) as estimated by the coefficient learning program PRDLRN, discussed below.

The "best" subgoal is selected so as to maximize the Slagle coefficient (1964):

$$\frac{\text{GPR (B)}}{\text{GC (B)}}$$

In actuality, this subgoal may not really prove to be the best one, if, say, the probability or cost estimates turn out to be incorrect. The performance program is a heuristic program and is forced to make decisions based on imperfect evidence.

A more sophisticated program would take into account the possibility of trying to achieve A, failing, then trying to achieve A' and succeeding. Thus a goal with several good

subgoals would have a larger GPR for this reason. However, INSIM1 considers only the one best subgoal.

(3) Suppose that A1 and A2 are components of the ordered-AND goal A1TH2 (A1 then A2). Then GPR (A1TH2) = GPR (A1) * GPR (A2).

(4) Notwithstanding any of the above, if a goal has already been achieved, its GPR=1. A goal is defined as "already achieved" if the corresponding signal has occurred within the last five seconds.

Similarly, the GC (time delay) of a subgoal is defined recursively as follows:

(1) For a response variable, GC = 0.

(2) If A is the best of several OR'ed subgoals of B, then GC(B) = GC(A) + GPR(A) Delay (A→B).

(3) The GC of an ordered-AND goal A1THA2 (A1 then A2) is GC(A1THA2) = GC(A1) + GPR(A1)*GC(A2).

(4) Notwithstanding any of the above, the GC of a goal is 0 if the goal is already achieved (in the past five seconds).

To summarize, in the feasibility study phase, estimates are made of the success probability and time delay of each path to the main goal.

2. The choice phase

The next step is to activate the goal tree branch which is estimated, according to simple heuristics, to be the quickest and surest path to the main goal. A goal is active if, and only if, its WANT variable has the value TRUE.

3. Computation of the WANT variables

(On a first reading, readers may skip to section 4 on the inner loop.) The WANT variable of a goal G is defined recursively as follows:

(1) If G is a main goal, WANT(G) = TRUE or FALSE as set by the motivation system.

(2) If A is one of several OR'ed subgoals of B, WANT(A) = (WANT B) $\bigwedge$ (A is not already achieved) $\bigwedge$ (A is the best subgoal of B)) $\bigvee$ (A is a curiosity goal (see below)), where "already achieved" means that the signal A has occurred in the last five seconds, and the "best" subgoal is that which maximizes

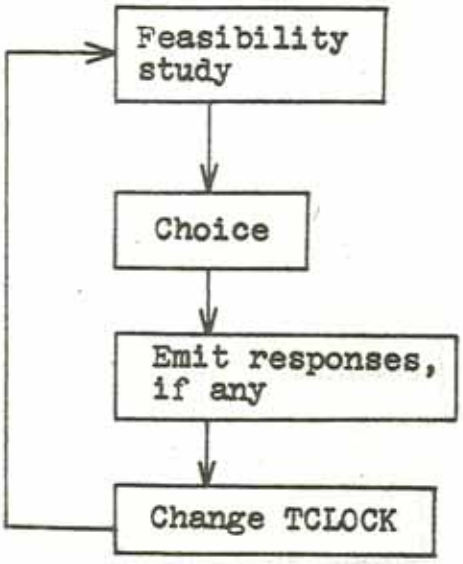
$$\frac{GPR(B)}{GC(B)}$$

(3) If A1THA2 is the ordered-AND subgoal "A1 then A2", WANT (A1) = WANT (A1THA2) $\bigwedge$ (A1 is not already achieved). WANT (A2) = WANT (A1THA2) $\bigwedge$ (A1 is achieved) $\bigwedge$ (A2 is not achieved).

(4) If G is a response (directly controllable) variable, WANT(G) causes the response to be emitted.

4. The inner loop

The feasibility study and choice phases are performed every time the simulated-time clock, TCLOCK, changes.



Thus the GPR, GC, and the program's decisions are constantly being updated on the basis of changing conditions. The PSIM interpreter ensures reasonable efficiency by recomputing only the variables which depend on some condition which has changed since the last TCLOCK time.

5. A sample problem

Now that the mathematics of the performance program has been presented in some detail, let us return to the thumb-sucking problem of section A and see how the mechanisms work in a concrete case. Assume the following values for conditional probabilities Pr (B | A)*:

Pr (face touch | lift hand) = 0.50
Pr (mouth touch | face touch) = 0.27
Pr (mouth touch | left cheek touch, then turn head left) = 0.78
Pr (mouth touch | right cheek touch, then turn head right) = 1.0

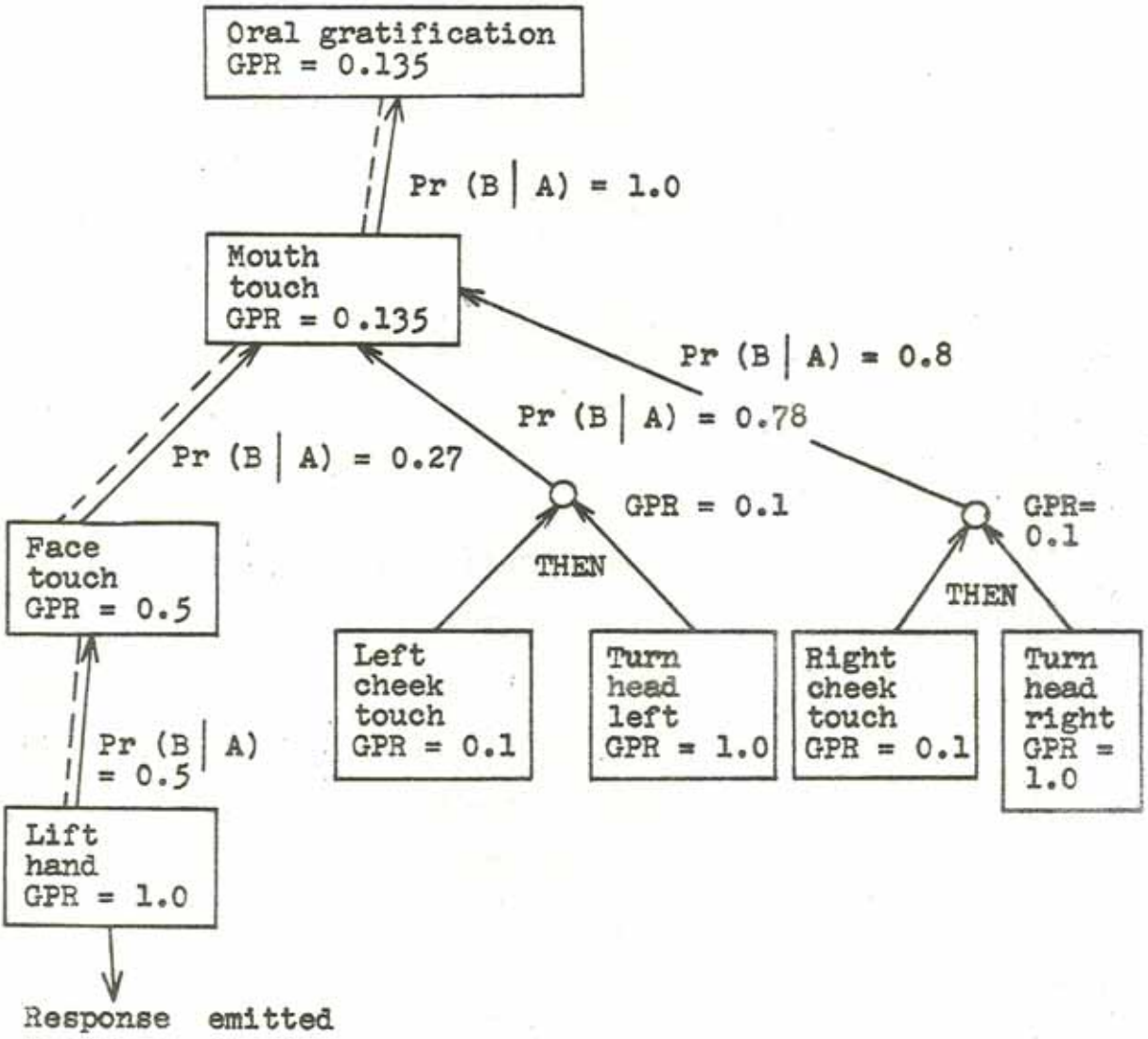
To simplify the discussion, assume that all time delay (cost) figures are similar enough that they do not affect the choices of which branch of the goal tree to activate.

Assume that the simulated infant has just come to want something in its mouth; i.e., that the motivation section has

---

*In some cases, these values are experimental results from actual runs on the INSIM1 program. In others, I have corrected for a bug in the program.

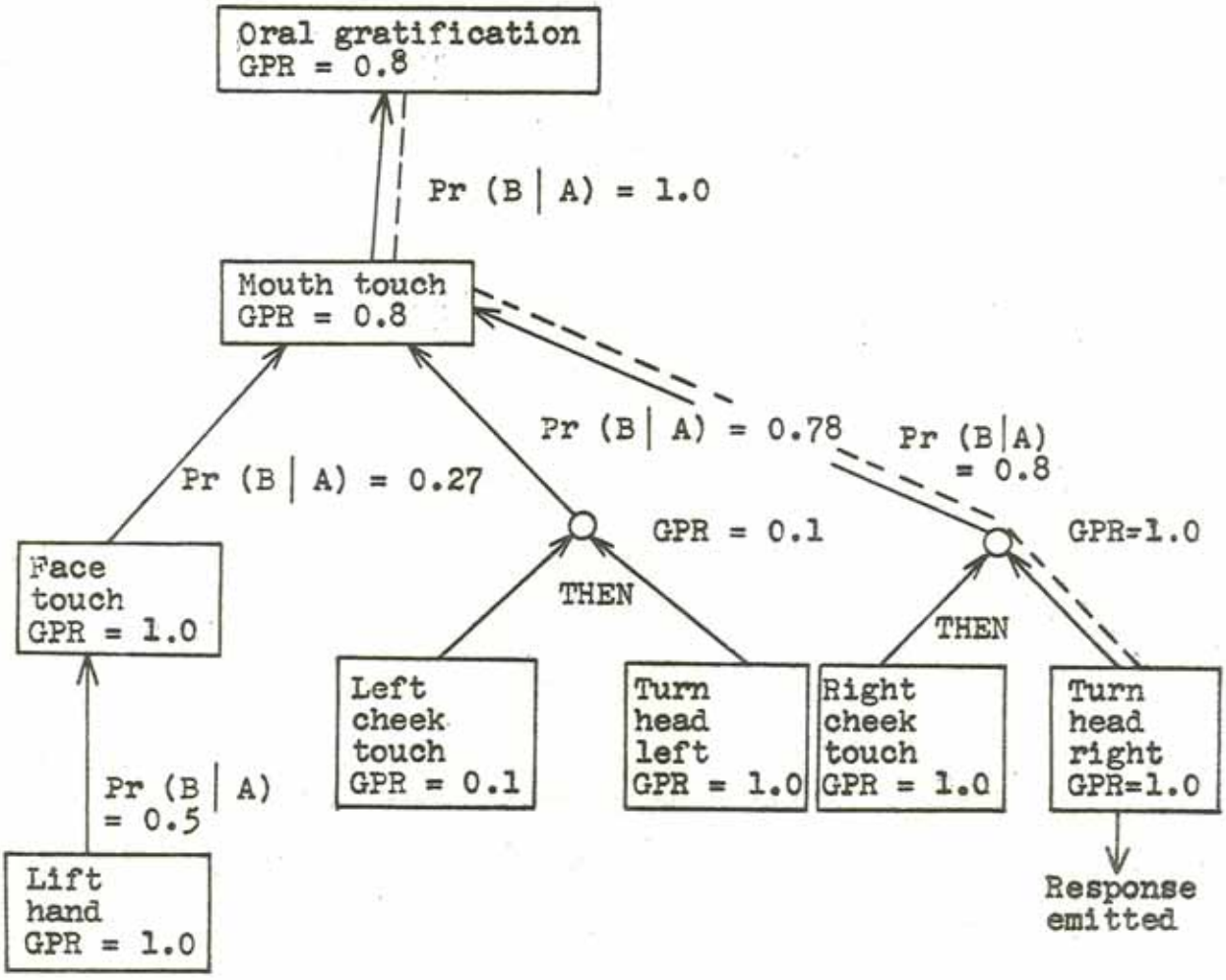
just set WANT (oral gratification) = TRUE. Also, assume that nothing has recently touched the infant's face or mouth. Under these conditions, the program will assign a low success probability, and hence a low merit, to the goal tree branches involving left cheek touch and right cheek touch. But the lift-hand-face-touch branch will have a higher merit and be activated.



The pulse on the "lift hand" line goes to the body and environment simulator, where a random number is generated and an outcome is determined. With probability 0.5, the hand misses the face. If it hits the face, it appears at the left cheek, right cheek, or mouth with equal probability.

Assume that the outcome is that the hand hits the right cheek. The body and environment simulator return a pulse on the "right cheek touch" line, and the display is updated to show the infant's hand at his right cheek.

("Right" relative to the infant; left relative to the page.) Next, the tree variables are updated to reflect the new situation. "Right cheek touch, then turn head right" now has a high achievement probability; its GPR = 1.0. Since it is strongly connected to "mouth touch," the branch ending in "turn head right" is activated.



The body and environment simulator receives the "turn head right" command, and, after a delay of 2 simulated-time seconds, sets the head position to "right" and returns a "mouth touch" pulse.

PRDLRN then increments Pr (mouth touch | right cheek touch, then turn head right) from 0.80 to 0.82.

6. Discussion

INSIM1 performance programs incorporate simple heuristics which work well in cases where the assumptions on which they are based hold hold true.

Among the assumptions are:

(1) Success probabilities and time delays are assumed to be statistically independent. If this is not true, the chaining formulas used in computing success probabilities and time delays will not be accurate.

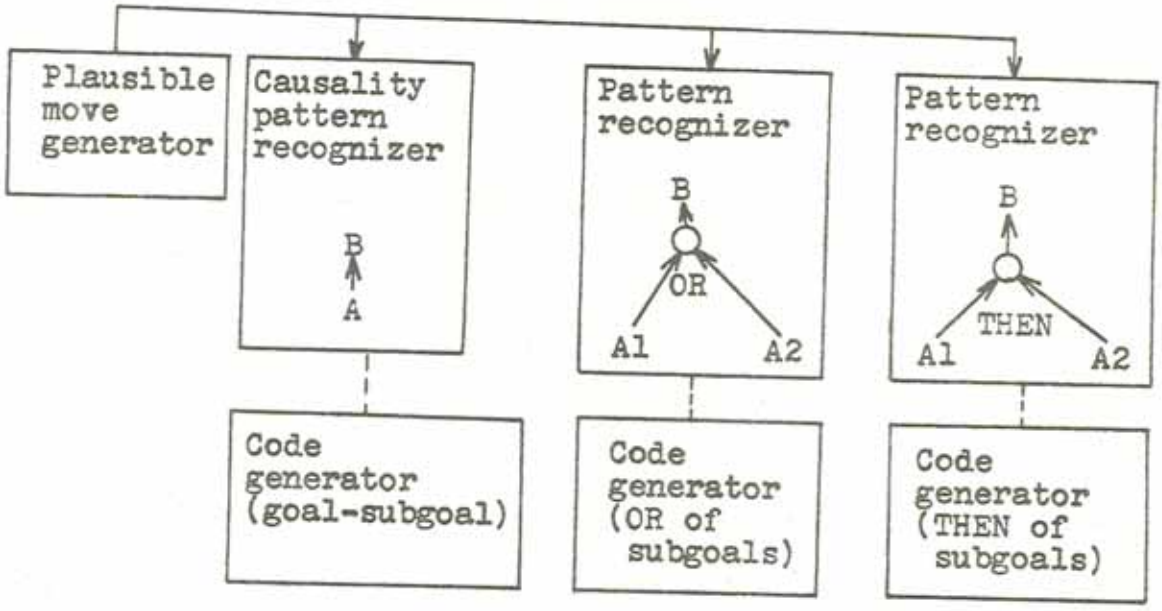
(2) It is assumed that goals do not conflict: i.e., that the achievement of one goal does not decrease the probability of achieving another goal.

Removing these performance limitations would require additional machinery beyond the scope of the INSIM1 project, such as a look-ahead method of the type used in chess programs.

C. The experience-driven compiler

As mentioned previously, most of the performance program is coded by an internal compiler which, instead of using as its input a source code prepared by a human, is controlled by the experience acquired by the program as it interacts with its (simulated) environment. In keeping with the dictum that in order to learn something, one must know something already, the compiler incorporates the probability formulas described above, plus knowledge of basic aspects of the physical world, including time and causality.

The compiler consists of pattern recognizers, code generators, and a plausible move generator (not implemented at this writing).

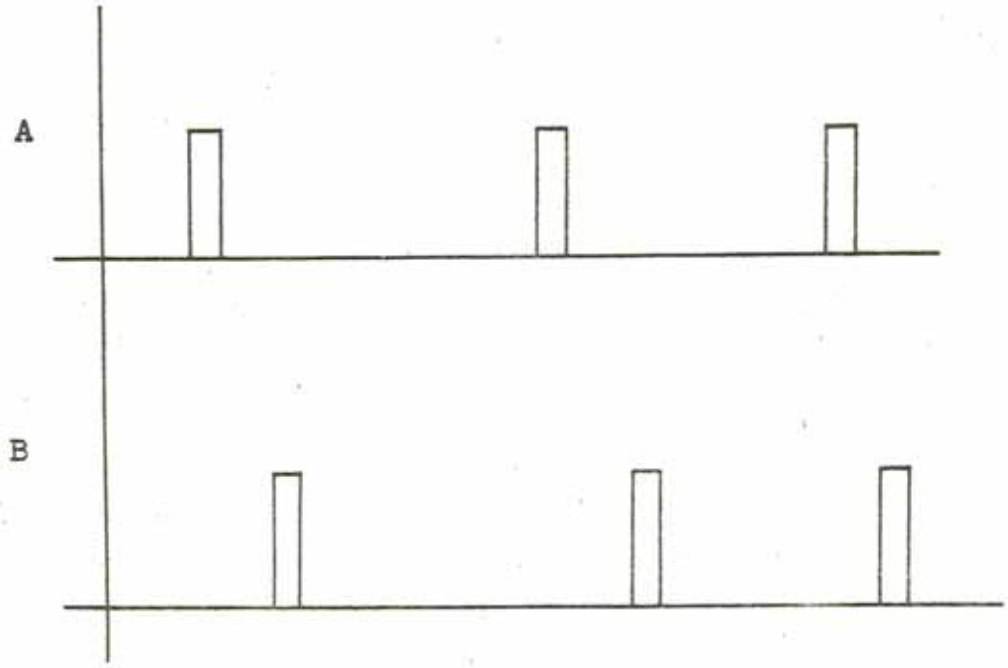


The plausible move generator is used instead of testing for causality between all possible variables A, B. The latter approach would involve on the order of n^2 tests, where n is the number of variables.

It is the compiler which sets the upper limit on the program's ability to learn. For example, INSIM1 could never learn to play chess even with very long training, because the necessary pattern recognizers and code generators are not present.

The experience-driven compiler operates as follows: The program starts out with an innate main goal which is "oral gratification" in the thumb-sucking problem. First, the plausible move generator is called to generate a list of variables which are likely to be "relevant" to the oral gratification goals, and causality test links (indicated by dotted connections) are formed.

Next, the causality pattern recognizer learns which test links represent actual causal relationships. The pattern it is looking for is:



If a pulse on variable A is followed by a pulse on variable B sufficiently often, A is assumed to cause B. More precisely, if $Pr(B \mid A) - Pr(B \mid A \text{ or } \sim A) > 0.15$ after at least 15 pulses on A and 15 on B have occurred, A is assumed to cause B. The pulses on A and B must be less than five simulated-time seconds apart. (If there are any pulses at all on B, then a pulse on A will always be "followed" by a pulse on B if we wait sufficiently long.) $Pr(B \mid A)$ is estimated by the coefficient learning program PRDLRN, discussed below.

These simple heuristics will miss some actual causal relationships when the delay is more than five seconds. It would not be hard to make the program "adaptive" to this arbitrary parameter, say by matching the time delay to the recent density of pulses on the two lines. Thus, if there were only one pulse on B about every 15 minutes, the allowable delay might be five minutes rather than five seconds. Also, the heuristics will sometimes "identify" a causal relationship where none exists. E.g., if the allowable time delay were long enough, it would think that day causes night. (Piaget has found that small children also think that day causes night.)

In some cases, it is sufficient to wait passively for a pulse on A. In other cases, the curiosity section of the

performance program sets WANT (A) to TRUE, activating some goal tree branch ending in A and initiating behavior which hopefully will lead to a pulse on A, in order to see if B follows (e.g., it activates "turn head left" to see if "mouth touch" follows); this is the "play" or "exploratory behavior" mentioned above. The curiosity section attempts to test links which are new and have not been tested many times; links where the initial variable, A, is reasonably easy to obtain; and where the final variable, B, is "biologically useful" (if one may use the term to describe a computer program) in that ability to obtain B would contribute to the program's ability to obtain primary reward. Specifically, the curiosity section tests the link A$\rightarrow$B which maximizes

$$\frac{GPR(A)}{GC(A)} \text{ Satfunc(A,B) Need(B)}$$

where Satfunc(A,B) (saturation function) decreases linearly from 1 to 0 as the number of times when A,B has been tested increases from 0 to 15. Need(B) is an index of how much the ability of the program to obtain primary reward would be improved by improvements in its ability to obtain B. See appendix 2 for a description of the heuristics used in computing Need(B). Only links A$\rightarrow$B which have been designated as "plausible" by the plausible move generator are tested, prevening an n^2 explosion as the number of variables increases.

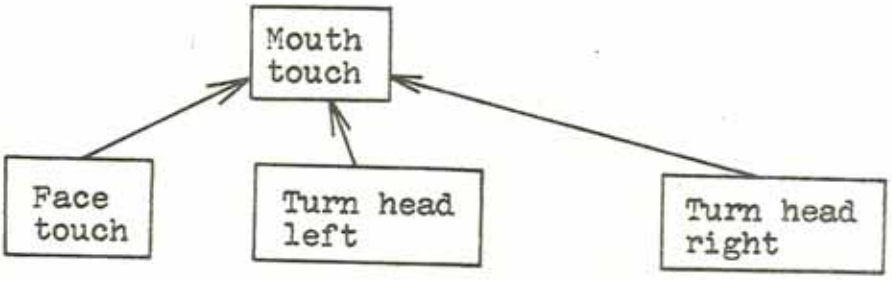
When the causality pattern recognizer detects that two variables, A and B, are causally related, the corresponding code generator is called to compile the link A→B in the goal tree. This code generator is a LISP function called MAKEORGOAL (A,B) so named because it also handles the case where A is one of several logically OR'ed goals. In LISP, the code generator turns out to be a straightforward and rather prosaic, if slightly long, program. Separate sections are provided for compiling the entries for WANT, GPR, GC, and each variable associated with the curiosity system. Each section looks up the names of the variables involved in the formula in question and substitutes them into the formula, using LISP's symbol-substituting capability.

Ordinarily, one considers it easier to write an interpreter for a particular language than a compiler for it. It would be possible in principle to store the goal tree in a very compact form as a set of links in storage, with a vector for each node to store GPR, GC, WANT, etc., then write an interpreter which incorporates the GPR, GC, and WANT formulas. The difficulty is that, somewhat counter-intuitively, the interpreter would seem to be quite a lot harder to write than the compiler. The most obvious algorithm for the interpreter would have a recursive function FINDR (find-response) which

would be called for each response variable each time the situation changes. FINDR would call the function FINDWANT to compute WANT variables; then FINDGPR and FINDGC would be called. But this algorithm would collapse if the goal "tree," rather than being a true tree, has loops in it. In this case, the algorithm becomes non-terminating, basically because the GPR-GC formulas then define a set of simultaneous equations rather than a recursive computation. In INSIM1, this problem is solved by PSIM, which is equipped to solve simultaneous equations when these occur.

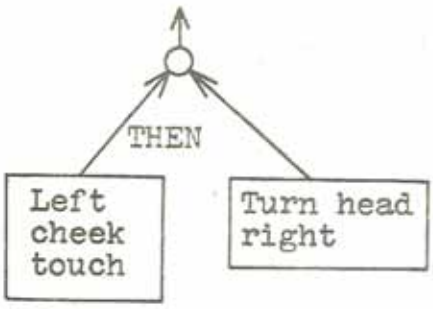
The only algorithm I have thought of which would work in a goal tree interpreter would recompute every variable of each goal, then recompute the variables of all goals linked to goals where a variable has changed, then iterate until no variables change. This algorithm is quite "brute-force," and I think it would show severe performance problems.

In the thumb-sucking problem, the program first learns the links:

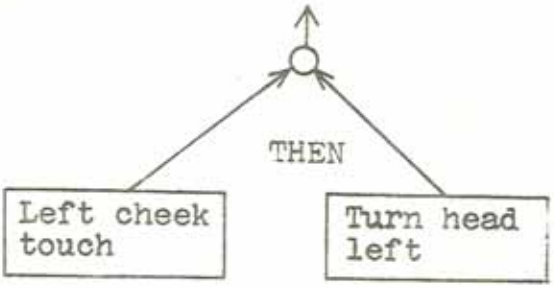


Although this version of the performance program will sometimes succeed in obtaining "mouth touch," it does not yet know which way to turn the (simulated) head.

Next, the plausible move generator is called to provide a list of variables to be THEN'ed with the partially successful subgoals. Causality test links are compiled for the ordered-AND variables. Among them are:



V1



V2

V1 correlates very poorly with mouth touch; V2 correlates very well. Since Pr (mouth touch | V2) is very high, the performance program will activate this branch, rather than the others, and the simulated infant will emit "turn head left" in response to "left cheek touch." Similarly, it learns to emit "turn head right" in response to "right cheek touch."

What is happening here is that the conditional probability figures, such as Pr (mouth touch | turn head left) are being used as a hill-climbing criterion in program space (Minsky, 1961, p. 10). (Turn head left | mouth touch) works

some of the time; INSIM1 forms new properties of the problem by combining properties which have proved useful in the past (Minsky, 1961, p. 13).

Finally, "face touch" is identified as a "biologically useful" variable, and the program learns to activate "lift hand"; when the (simulated) hand touches the face, the previously learned program takes over and completes the thumb-sucking operation.

One way of looking at the learning process is that the program builds a subroutine hierarchy. Each node on the goal tree defines a subroutine: the process of achieving a stimulus on the line defined by the node; e.g., the "obtain mouth touch" subroutine. Each link on the tree defines a subroutine call. Thus, the "obtain face touch" subroutine calls the "lifthand" subroutine.

See appendix 4 for a discussion of how the experience-driven compiler is organized and programmed.

It is interesting to note the similarity between this learning sequence and Piaget's observations on the learning of human infants. Although the real infant's learning is much more complicated, it follows the same gross sequence of stages; the real infant first learns to search from left to right with its head; then it learns which way to turn; then it learns to lift its hand and suck its thumb.

The next two sections are devoted to a discussion of the PSIM interpreter and the PRDLRN coefficient-learning program; they may be skipped on a first reading.

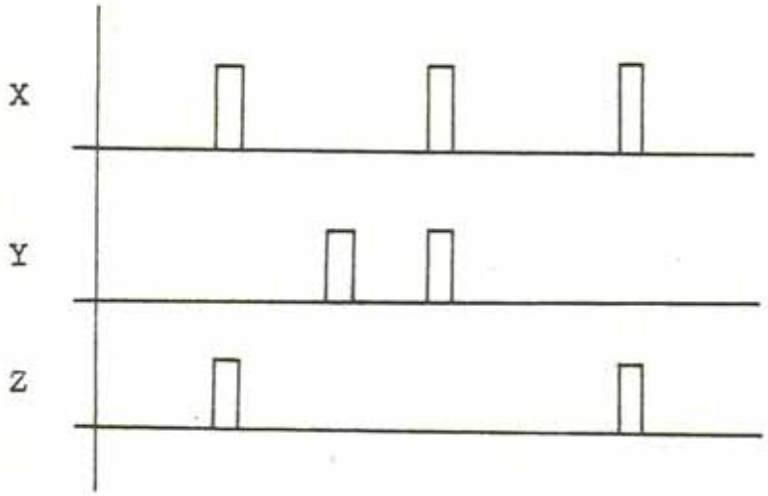
D. PSIM (Parallel Simulator)

The PSIM interpreter, embedded within LISP, handles the details of arranging the second-by-second occurrence of simulated events and relieves the compiler of the need to schedule the sequence of computations. A PSIM program consists of a set of variables, each of which has an S-expression which determines its value. E.g.:

```
(Z (AND X (NOT Y)))
(X (POISSON 0.1))
(Y (POISSON 0.1))
```

The Poisson expressions generate POISSON-distributed pulse trains with mean frequency 0.1 pulses per second.

Whenever a variable, such as X, changes the variables which depend on it are automatically updated. A graph of X, Y, and Z versus simulated time will look something like this:



PSIM also handles the complications which arise when the goal tree is circular; in this case, an iteration procedure is used to calculate the GPR, GC, and WANT variables.

Some readers have complained that they have trouble grasping the structure of this program because of the parallel simulation. See appendix 3 for a more detailed discussion.

E. PRDLRN (Probability-Delay Learner)

Conditional probabilities and time delays are estimated by a rather orthodox coefficient learning procedure (Minsky, 1961). Suppose there is a link between A and B. Whenever A occurs, followed within five seconds by B, Pr (B | A) is incremented by an amount θ (1-old value of Pr (B | A)), and Delay (A→B) is incremented by θ (actual delay - old estimate of delay). If A occurs, but not B, Pr (B | A) is decremented by an amount θ (old value of Pr (B | A)) and Delay (A | B) is incremented by θ (5 seconds - old estimate of delay). It can be shown that this procedure gives an unbiased estimate of Pr (B | A) and Delay (A | B), with an exponential weighting such that old occurrences of A affect the estimates less than new ones. θ , the decay coefficient, is currently 0.1. The initial estimate of Pr (B | A) is obtained by observing the first 10 occurrences of A. Pr (B | A) is set to:

$$\frac{\text{number of A's followed by B's}}{\text{number of A's}}$$

Experimental results

At this writing (August 1970), the INSIM1 program still has residual bugs in it, but it will perform well enough to demonstrate the coefficient learning, causality detection, goal tree formation and operation, and stages of learning. A test run has five stages and lasts for roughly 1800 simulated-time seconds.

Stage I: 0-400 seconds

In its earliest stage, the simulated infant is entirely passive. It is fed by placing the bottle in its mouth. Internally, the goal tree consists of only one link, a causality test link between "mouth touch" and "oral gratification."

Stage II: 400-1000 seconds

By t = 100 seconds, PRDLRN has detected the causal relationship between "mouth touch" and "oral gratification," and this connection shifts from a test link to a goal tree link. The (dummy) plausible move generator lists seven plausible causes for "mouth touch," and test links are prepared. These are: "face touch," "turn head left,"

"turn head right," "kick right leg," "kick left leg," "wiggle right toe," and "wiggle left toe."

In the new stage, the curiosity system can activate the head motions, and one can see it turning its head back and forth on the display screen, although in a totally uncoordinated manner.

Stage III: 1000-1200 seconds

At t = 1000, the causal relationship "face touch" to "mouth touch" is detected. At the same time, test links to "mouth touch" are filed for "right cheek touch then turn head right" and "left cheek touch then turn head left." These links affect only the exploratory behavior.

Stage IV: 1200-1600 seconds

At t = 1200, the causal link "turn head left" to "mouth touch" is detected. Now the program is slightly less helpless than it was at first; whenever it wants oral gratification, it performs a means-end analysis which leads it to "turn head left," which, if this is not always successful behavior, is at least relevant to the task at hand.

Stage V: $t > 1600$ seconds

At this point, the program becomes able to turn its head to the left in response to a left cheek touch, and it could respond properly to a right cheek touch, were it not for a program bug. Another bug prevents the development of the "lifthand" response. It is believed that these bugs are minor in nature. Other, undetected bugs may also exist.

Chapter 4: Analysis

This chapter is devoted to a discussion of the fundamental scientific issues to which the INSIM1 research addresses itself, and attempts to place the work in the context of research on learning and in the context of other research on artificial intelligence. The fundamental issues to which the INSIM1 work addresses itself are:

(1) Can one make use of a relatively small amount of very general innate knowledge in order to obtain a much larger amount of specialized knowledge, and, if so, how?

(2) What should the innate knowledge be?

(3) How should the innate knowledge be incorporated into an information-processing system?

These issues are as old as epistomology itself, but the first really careful analyses were by Hume (1777) and Kant (1781). Hume took the position that the human mind was a "tabula rasa" (blank tablet) at birth and that all knowledge was acquired through the forming of associations (compare Hebb's synaptic connections). Kant, on the other hand, believed that the infant had a store of innate (categorical, or a priori) knowledge at birth and that this was necessary

to make the learning process work properly. The INSIM1 research supports Kant's viewpoint, not for abstruse philosophical reasons but from practical engineering experience with information processing systems of this type. INSIM1 is associationist ("turn head left" is associated with "mouth touch"); however, in order to make the association proceed properly, innate knowledge had to be incorporated into the learning program, knowledge that there are such things as causality and time, and that causally related events are likely to occur in close temporal sequence.

A question immediately arises as to just what is meant by statements such as "INSIM1 knows that there is such a thing as causality." The word "know" can be used in several senses. Obviously INSIM1 does not know about causality in the same sense that an adult knows about causality; the program cannot explain causality, cite examples, or answer questions about causes and effects; it has no verbal behavior at all. Instead, to say that INSIM1 knows about a certain type of causality is to say that the INSIM1 program is optimized to a universe in which certain types of causal relationships exist. Thus, in a universe where there were no such things as causality or time, or where cause and effect were always separated by hours or days, INSIM1 would not work properly. In other words, the innate knowledge of

INSIM1 is incorporated in the form of algorithms rather than as facts.

The learned knowledge is also incorporated into the system in the form of algorithms rather than facts. Thus, the final version of the performance program knows right from left in the sense that it can turn its head in the proper direction if stimulated; yet it has no verbal knowledge of space at all. Expressing knowledge as algorithms, as in program learning, is meritorious in that it is algorithms which we know best how to combine into complex integrated systems. Thus INSIM1 learns a nursing subroutine, then adds additional code to form a thumb-sucking subroutine. By contrast, for a machine to learn facts is at present often like adding more books to a library; the machine cannot do much with them. Many of the theorem-proving efforts suffer from this problem. Expressing knowledge as facts has its complementary merits, as discussed by Hewitt (1969) and the present author (1966).

Given the concept of using a base of very general innate knowledge to obtain a much larger repertoire of learned knowledge, what can we say about what the innate knowledge (innate algorithm) should be; in particular, how much innate knowledge is needed and how problem-specific the innate knowledge needs to be? Why was the innate knowledge basis of INSIM1 chosen the way it was? For several reasons:

(1) The INSIM1 program has a high "bootstrapping leverage" ratio:

$$\frac{\text{Amount of learned problem solving ability}}{\text{Amount of innate knowledge}}$$

where the "amount" is to be indexed by some criterion or other. Although this is not a fact which can be easily demonstrated now, since the learning is severely limited by the body and environment simulator, INSIM1 is capable in principle of synthesizing very large trees of OR'ed and THEN'ed goals (limited by core storage), implementing long chains of behavior.

(2) INSIM1 is based upon a strong theory of problem-solving, means-end analysis (Newell, Simon, and Shaw, 1959), and one can be confident of the program's ability to be extended to other goal types and to solve harder problems.

(3) INSIM1 performance programs are free from the "exponential explosion" problem which plagues many problem-solving systems. The simulated time to learn a tree grows only linearly with the length of the tree; the CPU time per simulated time second grows a little worse than linearly with the number of branches on the tree, and the number of branches is limited very effectively by the causality pattern recognizer.

(4) Lastly, the INSIM1 setup was chosen because it works; i.e., it constitutes an integrated learning-behaving system. This requirement is perhaps harder to meet than appears on the surface; I have 1100 pages of notes on setups which did not work, developed before arriving at INSIM1. The unsuccessful efforts of Pavlov and Hull are also testimony to the difficulty of getting anything which will work at all.

An important issue in A.I. research is the relative amount of learned and innate knowledge which is necessary. One may place research efforts in a spectrum ranging from the Perceptron workers and some behaviorists on the environmentalist end of the spectrum, to the Greenblatt chess program (1967) on the nativist end. On this spectrum, INSIM1 is not far beyond the minimal self-organizing systems (see the papers in von Foerster and Zopf, 1962; and Yovitts and Cameron, 1960). I would like to make it clear that, to the extent INSIM1 is minimal, it is minimal because of the minimal resources of the author, and not because I think self-organizing systems should be or can successfully be minimal. See chapter 5 for an inventory of some of the innate implementation which I believe should be in a good learning code. Referring again to the ratio

$$\frac{\text{Amount of learned problem solving ability}}{\text{Amount of innate knowledge}}$$

I claim that:

(1) In a good learning code, the denominator should be much larger than in INSIM1; and

(2) Up to some high saturation point, the larger the denominator, the larger the ratio.

INSIM1 does not look very much like traditional artificial intelligence programs; it solves infantile rather than hard problems. (The learning program solves the hard problem of writing the performance program.) Nevertheless, it is indeed oriented toward similar goals and attitudes. It is an attack on the problem which occupied much of the science of psychology until the 1950's, the problem of getting a precise model of simple forms of learning, but now using the concepts and methods of modern computer science, with, I believe, much greater success than the earlier efforts. This line of research is, and will continue to be, heavily dependent on more traditional research in artificial intelligence.

INSIM1 is self-programming in the sense that, given some set of tasks, such as nursing from a bottle and sucking the thumb, instead of manually writing a performance program to perform the tasks, one has a learning program which writes the performance program on the basis of experience. The

dependence of this type of learning research on more traditional A.I. work lies in the fact that, to write the learning program, one must have a fairly clear idea of the structure of the performance program; e.g., that it is to be a certain type of goal tree. To coin a slogan, in order to write a program which prepares a program to do X, one must first be able to write a program to do X. The advantages of automating the process are the usual ones of greatly improved accuracy and potentially great saving in time and effort.

What are the prospects for using general learning techniques in practical computer programming? Certainly history gives grounds for nothing but pessimism. The best chess program, the best symbolic integration program, and the best vision programs do no learning at all. I am cautiously optimistic about the prospects for using the "experience-driven compiler" concept in certain application areas where the problem is fairly "sensorimotor;" i.e., not too different from what the infant learns to do, such as vision and hand control. Even INSIM1 could be interfaced quite readily into a robot to do a three-point block-moving problem, given LISP or assembly language primitive for "move block left," etc. The trouble with self-organizing vision is that we do

not know enough about what the performance program should look like. I am quite pessimistic about developing, in the near future, experience-driven compilers for adult-level problems such as symbolic integration.

One possibility is that this type of learning research may provide a psychologically liberating atmosphere in which to study problems such as computer vision. As Professor Joel Moses is fond of pointing out, intelligence is a "kludge," and good vision performance programs are likely to be as much composed of messy-looking collections of special-case tricks as orderly general principles. Vision researchers are often psychologically very uncomfortable with this, since they, like the rest of us, want their work to be general in its applicability. The experience-driven compiler provides a possible way out of this bind, since the learning program is quite general, specialized only to very broad aspects of the physical world such as time and causality, while the performance program contains the learned special-case tricks.

Some readers may wonder if the use of "high-powered" techniques such as end-means analysis and Slagle coefficients is warranted in modeling simple infant behavior. My experience has been that a fairly "rugged" problem-solving system is needed in a learning program where the problem solver

cannot be "spoon-fed" by a clever programmer. Something like the Slagle coefficients seems to be needed to get the program to activate the correct branch of the goal tree in a situation where many unsound branches may have been constructed by the learning program.* It may be true that simpler ways of expressing the performance program, such as a state-action table, would work in an environment as simple as the one described here. However, one wonders how much growth capability is available ultimately, without some sort of end-means analysis. It seems to me that end-means analysis, in the form of some sort of goal tree, is uniquely matched to the physics of the situation in which the infant finds himself. In the real world, there really are causes and effects; the causes precede the effects in time, and the effects in turn become new causes. We really do want to have some coded description of a desired end state (goal) and try to find a sequence of events (a plan) which will lead to achieving the goal. Each such event becomes a subgoal and is treated recursively in much the same way as a main goal, hence the goal tree.

---

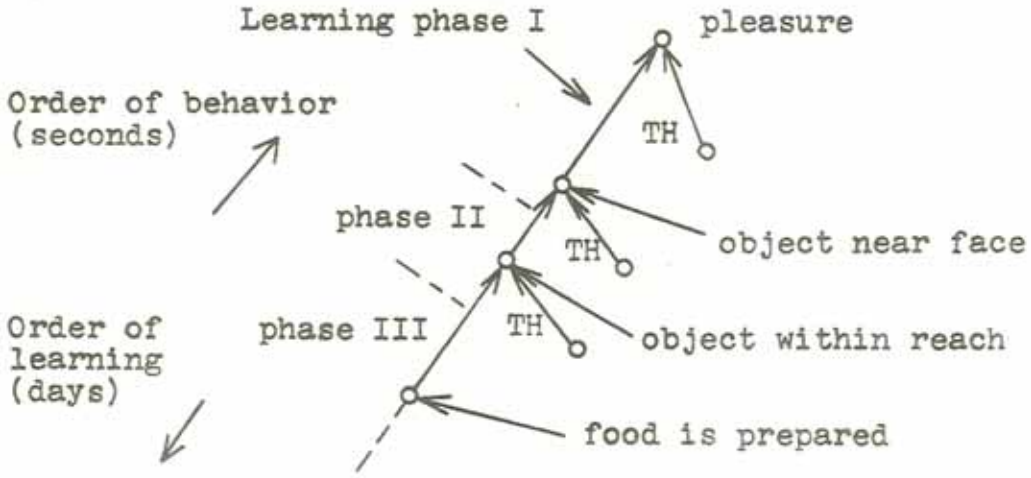
*Of course, since the Slagle coefficients are only heuristic approximations, and since one has great latitude in the details of end-means systems, I am not suggesting that exactly a certain computational form seems to be required, only a computational quality.

Perhaps the major unresolved issue in machine (or human) learning concerns the number and complexity of types of learning which are involved in intelligent behavior. The typical learning theory tries to synthesize everything out of one or sometimes two forms of learning, such as Pavlovian conditioning or trial-and-error learning. My feeling is that a fully developed learning system would be lucky to get by with less than fifty essentially distinct forms of learning. Even INSIM1, whose capabilities are quite limited compared with what will ultimately be needed, has two forms of learning (coefficient learning and program learning) with a third, learning to give up, to be added as soon as it can be programmed.

We may be in for some pleasant surprises if someone devises a very clever way of synthesizing many good behavior-defining functions from a smaller set of primitive functions. We simply do not know enough, now, to say with assurance that it is absolutely impossible for a program no larger than GPS to bootstrap its way into intelligence. In fact, a very small, inconceivably inefficient mechanism of the sort considered by Solomonoff (1957; 1960) for theoretical--indeed, philosophical--purposes could probably do it, but that is essentially another subject because it

involves such absurd eons of brute-force search. And what we know of genetics and other evidence of biological complexity, makes it seem not so urgent to look for mechanisms with that small an innate endowment.

INSIM1 goal trees contain a relatively high density of branches which actually contribute to the solution of the problem, compared with theorem-proving programs and chess programs, in which most of the tree branches turn out to be useless. The density of useful branches will be even higher in future versions of the program, which will be equipped to prune the tree of branches which are no longer used. In general, learning takes place as the program takes over more and more of the sequence of events leading up to reward. Thus, at first, the simulated infant's simulated mother must place the bottle in the infant's mouth. Then the simulated infant takes over the final stage in the sequence; getting its head into the proper position. Next, it becomes able to provide its own object (the thumb) instead of depending on its mother. One can imagine a future adult able to prepare the food or earn the money to buy it. In terms of the goal tree, the learning sequence is typically one of adding links in an order opposite to the order in which they are used in behavior:



Note the large fraction of tree branches which actually have a beneficial effect on behavior.

Mention should perhaps be made of two appealing formulations of the learning problem which did not seem to play a role in INSIM1. The first is the statement that, in order to be capable of learning something, a program must first be capable of being taught it. IMSIM1 is capable of being taught only in the sense that a complete programming language

is capable of being taught anything by writing the appropriate program. The second formulation is that the learning program needs to have a "model of itself." IMSIM1 does keep fairly detailed records about which pairs of variables are linked together, the names of various variables, etc. Only in this sense, and in the sense that a program is its own model, does INSIM1 have a model of itself.

Despite the name "infant simulator," only strictly limited claims are made about the degree of biological realism of INSIM1. Thus, when I say that INSIM1 has "innate" formulas for calculating GPR, I am not suggesting that the real infant has some representation of the formulas in his DNA code; only that the formal model behaves as if he did. Nothing is known about DNA coding of behavior-related information. At the current state of computer modeling, if one is interested in learning about real babies, one should study real babies rather than make computer models. Weizenbaum makes a distinction between "theory" mode, "performance" mode, and "simulation" mode. In theory mode, one is concerned with getting a good formal model of some system; in performance mode, the goal is to get a program which performs well on some interesting (preferably useful) set of problems; in simulation mode, the objective is to get a model which is realistic enough to, say, substitute

for actual observations of the system in question (such as a simulation of a space flight). The INSIM1 work is basically in theory mode; we get some performance out of it; but as for biological realism, all we can say is that it exhibits stages roughly analogous to those exhibited by a human infant.

I am indebted to Professor Marvin Minsky and Professor Seymour Papert for several ideas expressed in this chapter.

Chapter 5: Proposals for future research

This chapter has two purposes: One is to present proposals for computer programs which incorporate more advanced forms of learning than INSIM1; another is to suggest an answer to the fundamental question of how much and what kind of innate implementation is needed for a program to bootstrap itself into intelligence. This will be done by making an inventory of subsystems which I believe should be in an advanced learning code.

The proposals range from packages whose preliminary design is complete and which are ready to program, through concepts which are "half-baked" but in which we have a reasonable chance of achieving the goals, if perhaps not by the methods advocated, to areas, such as learned motivation, where one can only speculate and bemoan our ignorance of the subject.

A. The INSIM1 plausible move generator

Recall that, in Chapter 3, section C, mention was made of a not yet implemented plausible move generator which was to be called to make hypotheses about which variables might be causally related to some goal B. (If we tested all possible variable pairs A, B, the machine time needed would increase on the order of n^2

where n is the number of variables, constituting an intolerably large CPU and core storage load for large n.) The plausible move generator will use three relevance heuristics:

(1) (Implemented) Innately known relevance

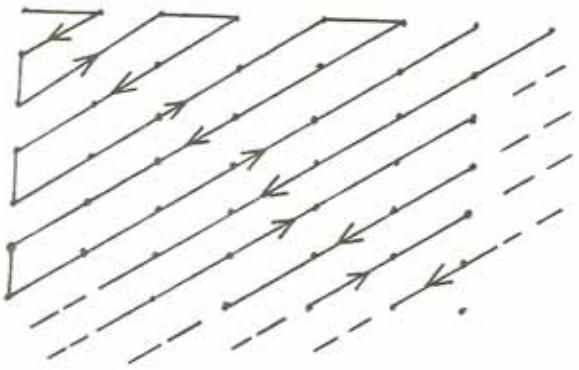
A variable A is innately known to be relevant to a variable B if A appears under B in the innately known file (i.e., manually prepared file) RELVLIST1.

(2) The diagonal search

This will be used to associate variables which are very "important", such as a "large moving visual stimulus" and "arm motion," and will be used to make the initial connections between sensory modalities and motor units such as arms or the head. The "important" variables will be placed on an innate list in decreasing order of importance. Let V_k be the $k^{\underline{th}}$most important variable. Make a square matrix:

$$\begin{array}{cccc} V_1\text{-}V_1 & V_1\text{-}V_2 & V_1\text{-}V_3 & V_1\text{-}V_4 \quad \ldots \\ V_2\text{-}V_1 & V_2\text{-}V_2 & V_2\text{-}V_3 & V_2\text{-}V_4 \quad \ldots \\ V_3\text{-}V_1 & V_3\text{-}V_2 & V_3\text{-}V_3 & V_3\text{-}V_4 \quad \ldots \\ V_4\text{-}V_1 & V_4\text{-}V_2 & V_4\text{-}V_3 & V_4\text{-}V_4 \quad \ldots \\ \vdots & \vdots & \vdots & \vdots \end{array}$$

Now let a dot represent a matrix entry and search the matrix in the order specified by the arrows:



Roughly, the concept is that the most important variables are associated with each other first.

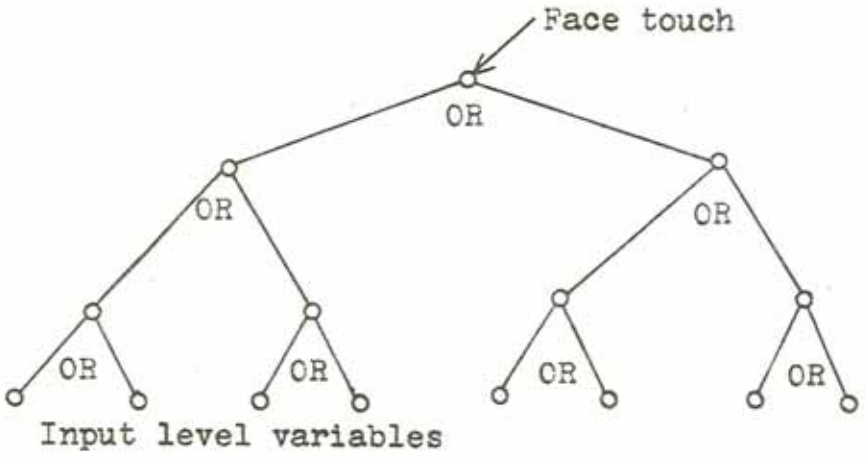
(3) The relevance chainer and the innate net

The innately known relevance subroutine and the diagonal search subroutine will be used to get the learning process started. To continue it, the relevance chainer section of the plausible move generator will be used. The relevance chainer will incorporate two heuristics:

1. If x is relevant to y, then y is relevant to x (symmetry).
2. If x is relevant to y, and y is relevant to z, then x is relevant to z (transitivity).

The relevance chainer will be used together with an innate net which will incorporate innate knowledge about which sensory modality a given signal belongs to and innate

information about space (which signals "belong together"). For a simple example, assume a one-dimensional space of touch receptors. The receptors will be arranged in an OR tree with a hierarchy of larger and larger sectors.



(Do not confuse this with a goal tree; the upper level variables are defined as OR's of the lower level variables.)

B. The binary valued goals package

At this point, the discussion shifts to the subjunctive mood ("would" rather than "will"), since the mathematics of this algorithm has not been completely worked out and no firm decision to implement it has been made. INSIM1 is restricted to pulse goals; i.e., goals of the form "cause a pulse to appear on variable V" (such as "mouth touch"). The binary valued goals package would extend INSIM1 to handle the case where a goal variable

could take a TRUE or FALSE value and hold it for a period of time. This is advantageous, since often we want a state of affairs to continue for a while ("something is in the mouth," rather than "something momentarily touches the mouth").

The binary variables package would include three goal types:

(1) "Achieve" goals

These goals are of the form "Set variable V to TRUE." An example would be "Set the mouth touch flag to TRUE."

(2) "Hold" goals

A "hold" goal would be of the form "Maintain the variable V, which is TRUE, for the next five seconds." A "hold" goal would not be WANT'ed unless the variable were already TRUE.

(3) "Avoid" goals

An "avoid" goal would have the form "Prevent the variable V from taking the value TRUE for the next five seconds."

A possible goal conflict could occur here. "Avoid" goals would have priority; i.e., if the "avoid" and "achieve" goals were both WANT'ed, the "avoid" would take effect and the GPR of the "achieve" goal would be set to zero.

The probability-delay learner, PRDLRN, would be extended to obtain probability and delay values for this case.

An example of a problem which could be solved by the binary valued goals package would be a problem in which a robot is to move a block back and forth between three points in one dimension (left, center, and right), with a binary variable for the gripper position. The program could learn to grip at the proper time, provided that having the gripper in the wrong position during a grasp operation did no permanent damage to the robot.

C. The continuous valued goals package

The learning codes discussed above share a common difficulty in dealing with problems relating to space: They are restricted to a few points (left, center, and right the current body and environment simulator) with the learning time increasing in proportion to the number of points. Thisdifficulty would be eliminated by a package which would equip the program to deal with continuous valued variables (e.g., "object x co-ordinate") in cases where simple hill-climbing will work. There would be two types of goals: "Increase (or decrease)x," and "set x=A." A fairly elaborate pattern recognizer and

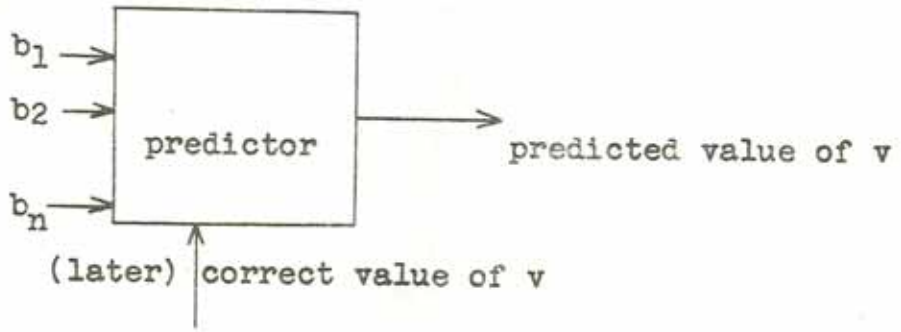
probability-delay estimator would be needed. A preliminary version of this program has been written by J. Kohr.

A very interesting problem is that of predicting the value of A such that some goal is achieved (e.g., the value of "hand x-coordinate" such that "hand touch" is achieved, in a grasping problem). This would be done by making a prediction of A, searching around with x until the goal is achieved, if possible, then comparing the predicted and correct values of A. Compare Samuel's (1959) process of comparing predicted and actual checker board evaluations. Also compare Halstead, Uber, and Gielow's program (1967). A polynomial of the Samuel type could be used for the prediction, perhaps with a technique to allow different polynomials to be used for different sectors of space. Since the co-ordinate transformation laws are fairly close to polynomials, the program could learn a very powerful technique, co-ordinate transformation, by a very general method.

If given the proper visual input, the program would be able to grasp a block, not just at three points, but at an arbitrary position.

Some readers may be interested in a more detailed description of how this could be done; others may skip to section D without loss of continuity. Formally, the

problem is what I have referred to in Chapter 2 as the signal prediction problem: We have a device which emits predictions about the value of some variable v (compare: some co-ordinate of an object to be grasped) in terms of the values of some set of basis variables b_1, b_2..., b_n (compare: visual variables related to the object). After each prediction, the correct value of v is fed back into the predictor and used to improve it.

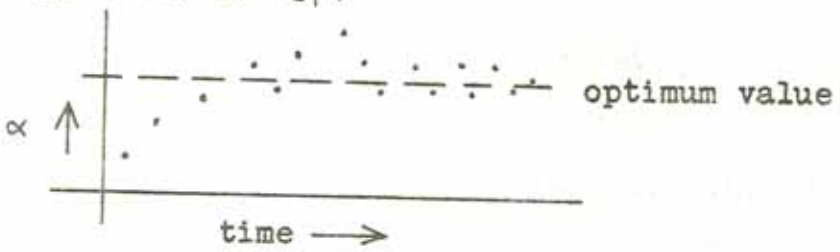


In order to improve the predictor, we will use two learning mechanisms: (1) a statistical coefficient learner to hill-climb on the existing formulas, and (2) a plausible move generator and predictor evaluator for symbolic learning. The coefficient learner would work as follows: Let α be a parameter involved in the prediction, and let ϵ_i be the $i^{\underline{th}}$ error term ($i^{\underline{th}}$ true value of v - $i^{\underline{th}}$ predicted value of v). Then, after each correction, change α by

$$\Delta\alpha = -\frac{\theta}{2}\frac{\partial}{\partial\alpha}\epsilon_i^2 = -\theta\,\epsilon_i\frac{\partial}{\partial\alpha}\epsilon_i$$

$$= \theta\,\epsilon_i\frac{\partial}{\partial\alpha}\left(i\text{th prediction}\right)$$

where θ is a rate constant; a large θ will mean faster learning but a less reliable answer. (See Samuel (1959) for a careful discussion of this type of coefficient learning technique.) For example, suppose the program thinks that v has the form $v = \alpha_0 + \alpha_1 b_1$. As learning proceeds, α_0 and α_1 will converge toward and then fluctuate about the "optimum" value, the value which minimizes the average value of ϵ_i^2.



The plausible move generator would be responsible for generating the formula in the first place. Initially, the formula would be v = a constant. Plausible moves would include:

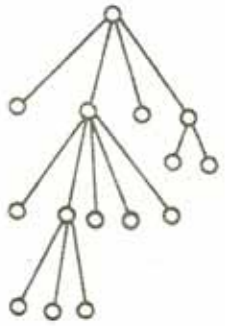
(1) Adding a new variable to the formula, with a constant multiplier. E.g., $v = \alpha_0$ becomes $v = \alpha_0 + \alpha_1 b_1$ with α_1 initially set to zero.

(2) Generating a term b_i^2, where b_i is already in the formula.

(3) Generating a term $b_i b_j$, where b_i and b_j are already in the formula.

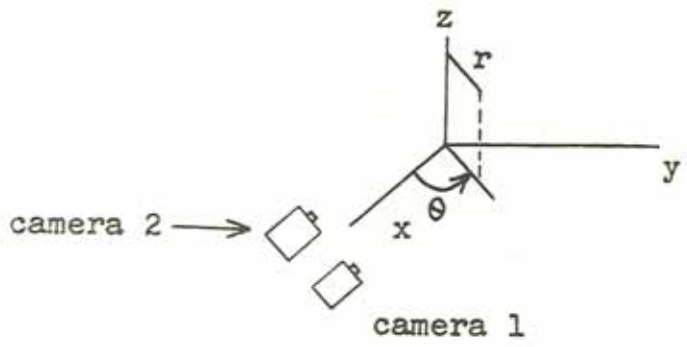
(4) Splitting the domain of the function into two sub-domains with separate coefficients.

Hopefully, it would be possible to restrict the search so that only one or a few of the plausible moves would need to be made the subject of a new plausible move generation.

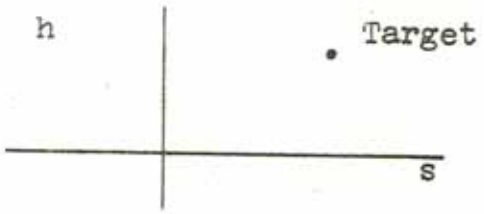


This would be done by restricting the search to one or two nodes having lower average E_i^2 than their competitors. Thus E_i^2 would be used as a hill-climbing criterion.

To illustrate the operation of the numerical variable predictor, assume a situation in which an arm is controlled by giving it the cylindrical co-ordinates r, θ, and z of the point to which the arm is to move.



Two TV cameras pointing down the axis provide stereoscopic information about the location of a target object.



Let h, s_1, and s_2 refer to the vertical and horizontal locations of the target as seen by camera 1 and camera 2. If the cameras are reasonably far from the scene, we may make the approximations that s, y, and z take the form:

$$x \cong C_{sy} \frac{1}{s_2 - s_1} + D_{sy}$$

$$y \cong A_{sx}\, s + B_{sx}$$

$$z \cong A_{hz}\, h + B_{hz}$$

where $s = \frac{s_1 + s_2}{2}$ and the capitalized symbols are constants. The transformation laws we are looking for take the form:

$$(1)\quad z = A_{hz}\, h + B_{hz}$$

$$(2)\quad R = \sqrt{(A_{sx}\, s + B_{sx})^2 \quad + \left[C_{sy} \frac{1}{s_2 - s_1} + D_{sy}\right]^2}$$

$$(3)\quad \theta = \sin^{-1} \frac{y}{R} = \left[\frac{1}{R(h,\, s_1,\, s_2)} \quad (A_{sx}\, s + B_{sx})\right]$$

One can trace the stages in the learning of these laws as follows:

Phase (1): The program starts out with the initial approximation z = a constant.

Phase (2): The plausible move generator guesses that z might take the form $z = \alpha_1 h + \alpha_0$. After the coefficients are learned, this equation turns out to predict z with excellent accuracy, due to its resemblance to equation (1).

Phase (3): After trying R = a constant, the program then tries:

$$R = \alpha_0 + \alpha_1\, s$$

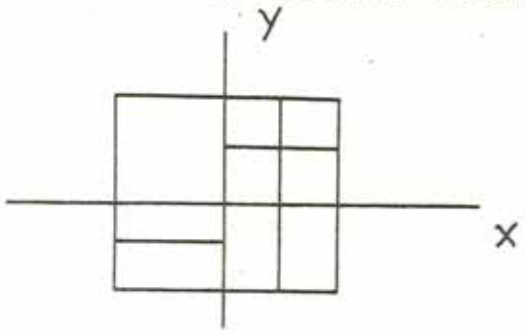
$$R = \alpha_0 + \alpha_1\, s + \alpha_2\, s_1$$

$$(4)\quad R = \alpha_0 + \alpha_1\, s + \alpha_2\, s_1 + \alpha_3\, s_2$$

along with other possible variables.

Phase (4): After equation (4), which has all the relevant variables in it, is discovered, the domain of the function is subdivided and piecewise linear or quadratic curve fits are made for each subdivision. This process may be

illustrated in a two-dimensional domain.

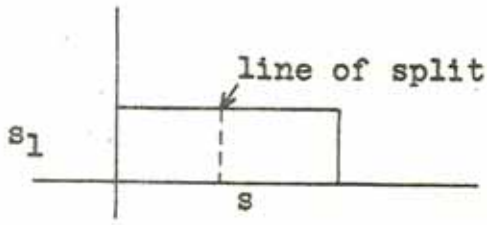


A simple, but serviceable, way of doing the domain splitting is the following: Suppose we are trying to make a fit of the form $R \cong \alpha_0 + \alpha_1 s + \alpha_2 s_1$ and suppose the linear function gives a poor fit in some domain. The plausible move generator would try splitting on S, S_1 and S_2. Let us look at the process of splitting on S in some detail. Suppose the original domain was

$$A_1 \leq s < B_1$$

$$A_2 \leq s_1 < B_2$$

A simple way to split on S would be to simply split the domain down the middle:



Left subdomain: $A_1 \leq s < \frac{A_1 + B_1}{2}$

Right subdomain: $\frac{A_1 + B_1}{2} \leq s < B_1$

Next, the split would be evaluated by making separate curve fits on the subdomains and testing to see if the fits are "substantially" better than on the original domain, i.e., if the split leads to "substantially" less noisy predictions. If so, the new domains would be further subdivided as needed.

An alternate way of doing the domain splitting would be to use a linear separation function of the form $as + bs > \theta$, obtimizing the coefficients a and b. While getting from (4) $R = \alpha_0 + \alpha_1 s + \alpha_2 s_1 + \alpha_3 s_2$ to an approximation of

$$R = \sqrt{(A_{sx} s + B_{sx})^2 + \left[C_{sy} \frac{1}{s_2 - s_1} + D_{sy}\right]}$$

may seem quite a jump, the hard part is getting equation (4).

The predictor for Θ would be learned in a similar way.

Because of the domain splitting, this algorithm exhibits an exponential explosion as the number of variables increases. One vitally needed ability is that of making compositions of functions previously discovered to be useful (McCarthy, 1956). E.g., after the R function is learned, it would be used to form the Θ function.

The learning process would be speeded up if the plausible move generator created terms of the form $\sqrt{b_1^2 + b_2^2}$ on the ground that adding intensities often makes sense.

D. The tree command language and associated software

As mentioned in chapter III, INSIM1 performance programs have "bugs" in cases where goals conflict. The binary variables package will have similar problems in cases where the variables on which GPR predictions are based change with time. At this point, the language shifts from "would be" to "might be," since I cannot claim to know how to solve these difficult problems. Since it is clear that they need solving, the tree command language and its

associated software are listed here as part of the inventory of innate implementation which I think belongs in an advanced learning program. Recall that INSIM1 performance programs consist of a goal tree whose branches are activated when the WANT variables of the appropriate sub-goals are set to TRUE. The tree command language (abbreviated T.C.L.) is a way of expressing commands to activate branches under certain conditions where a more sophisticated algorithm is needed than the goal tree itself provides. An example is (IMPLIES FACETCH MOUTCHWANT) meaning "If 'face touch' occurs, activate the goal 'mouth touch.'" In psychological terms, the T.C.L. system corresponds more or less to habit formation.

A typical T.C.L. setup might look something like this:
(1) If G is a goal and GWANT is its WANT variable, GWANT is a T.C.L. program meaning "activate the goal G."
(2) If P is a T.C.L. program and V is a variable, (IMPLIES V P) is a T.C.L. program meaning "If V occurs, activate the program P."
(3) If P_1, $P_2 \ldots P_n$ are T.C.L. programs, then (SEQUENCE P_1 $P_2 \ldots P_n$) is a T.C.L. program meaning "Activate P_1. When its goal(s) are achieved, activate P_2, then the others in sequence through P_n."

(4) If P_1, $P_2 \ldots P_n$ are T.C.L. programs, then (SIMUL P_1 $P_2 \ldots$ P_n) is a T.C.L. program meaning "Activate P_1, $P_2, \ldots P_n$ simultaneously."

T.C.L. programs would be read and commands issued by the T.C.L. interpreter, which should pose no special problems. The programs would be generated by a plausible program generator and tested by a program evaluator. The plausible program generator would incorporate various strategies for avoiding goal conflicts. Suppose, for example, that A and B were binary variables with the goal A AND B. The goal tree first tries to achieve A, then B. But suppose that achieving B sets A back to FALSE. The program might try:

(1) Achieving B first, then A.

(2) Getting B past the step which interferes with A, achieving A, then restarting on B.

(3) Getting A up to the stage where it would be interfered with by B, then achieving B, then restarting A.

The program evaluator would select the best of the plausible programs. It might operate by a controlled randomized experiment in which the learning program "tosses a coin" to decide which T.C.L. program to use, keeps records on success and failure, and finally adopts the best T.C.L. program.

The weak link in the system is the plausible program generator. It might very well generate an exponential explosion of bad T.C.L. programs unless designed with great care.

This proposal recalls behaviorist learning theory (Thorndike, 1898; Skinner, 1953).

E. Innate problem-specific software

While this thesis presents a "learning-oriented" approach to the artificial intelligence problem, one must not fall into the trap of a naive environmentalism and ignore the possible role of old-fashioned, manually prepared, innate problem-specific software--as opposed to the learning software, which is very general and problem-independent. For example, an organism that is to see and move would be immensely better off with elaborate special software for such things as visual pre-processing and motor control. The question of just what should be innate and what should be learned in models of human infants is one which can be settled only by prolonged development of and experimentation with learning codes. A good heuristic is that capabilities which seem to be innate in infants should probably be innate in our first general learning programs and capabilities which seem to be learned should

be learned. (Most capabilities are not known to be one way or the other.)

In studying the literature on lower animals, one is struck by the richness and complexity of the innate software. Nest-building behavior in birds develops perfectly well when the bird is raised in isolation and never has a chance to learn nest-building by watching another member of the species. To be sure, some kind of learning may well be involved; the bird probably builds a better nest on the second try.

Hubel and Wiesel have found that the lowest-level phases of vision, involving perception of edges, spots, and corner, are innate in the cat. In the case of humans, Gesell's (1948) observations suggest that several things we ordinarily think of as "learned" in the child seem to be largely innate, such as walking. Babies who accidentally break a leg and are immobilized become able to walk at about the same age as healthy children. Apparently the appearance of walking behavior is due to the maturation of structures in the child's nervous system.*

---

*But, of course, it is hard to separate this idea from the view that at least part of walking is learned, usually rather quickly once the necessary neural pathways become available, and could not be learned before that for the same reason.

In the case of machines, the most obvious place where a good innate implementation is needed is to provide useful sets of innate feature-detectors for situations or objects. In INSIM1, things are carefully arranged so the innate features, such as "left cheek touch" and "face touch," correspond well to what the machine needs to solve the problems we give it. EPAM would not work well if the tests in its tree were such things as the number of zero crossings made by the (cursively written) characters in the syllable; instead, EPAM's tests are, in some sense, "useful" features of the syllable.

F. One-trial learning capability

Up to now, the inventory has included only learning methods involving slow, patient statistical learning. But the vast majority of facts that we know are learned as the result of only one or a few experiences. After being introduced to someone, one can remember the person's name after being told it only once. (And forget it with equal ease, but that is a different matter.) EPAM (1963) is the protypical one-trial learning system.

G. Look-ahead capability

The mechanisms described previously would have no ability to make logical deductions and store the results or

to test a plan "in simulation" before trying it. A good learning code must be able to do these things. "Look-ahead" is a term broad enough to cover such types of information processing; hence the term "look-ahead capability."

Skinner (1957) has made the penetrating comment that thinking is covert behavior. We may expect to find the same mechanisms which are used to optimize other forms of behavior used to control the look-ahead process. For example, the look-ahead system might use goals of the form "Reduce the uncertainty about whether goal G can be achieved," with the elementary continuous-valued goals package used to learn the information needed to control the much more advanced look-ahead mechanism. Compare Newell and Simon's (1960) plan to use GPS recursively to control its own learning process.

H. Goal tree generalization capability

Another capability needed is that of generalizing from a pattern of goal tree entries. For example, let T_i be the i^{th} face touch receptor in a one-dimensional problem, and let THR be a response which moves the head so that an object moves from the i^{th} to the $i+1^{th}$ receptor. The goal tree pattern recognizer would discover this empirically, abstracting the fact that (T_i THEN THR) T_{i+1}.

The abstracted equation would then be used to compile many tree links. By contrast, INSIM1 would have to learn each link separately. The ability to make this type of symbolic abstraction seems to be an important component on intelligent thought. Compare Evans (1964). Here is also a case where the program should recursively control its own learning.

It is not absolutely impossible that a very clever abstracting program could be developed which would take over a large fraction of the work which I have assigned to the goal tree mechanism, the entity recognizer, etc. This program would synthesize performance programs, try them out, and keep the ones which lead to the most reward. It would have to be smart enough to invent the goal tree concept for itself. (Compare Friedberg (1958).) Until this technology appears, we should conservatively assume that it will not appear, and continue to develop such techniques as goal trees ourselves.

J. Learned motivations

Since the time of Freud, few subjects have received more scientific attention (or controversy) than that of how motives (compare "main goals") are or might be learned.

Nevertheless, this is an area where artificial intelligence theorists will have to start essentially afresh, since the current theories do not seem well enough structured to suggest details for computer implementation.

Although it is rather speculative to discuss the personality and motivational characteristics of an artificially intelligent being, so much has been said about the subject by science fiction writers, apparently with great influence on the general scientific community, that I cannot resist the desire to introduce some informed common sense. The term "robot" was invented by Karel Capek (1923) in his R.U.R. (Rossum's Universal Robots). Capek's robots were pseudo-biological slaves which were sold, presumably at a rather low cost, to do a variety of not very inspiring tasks ranging from factory worker to household servant to executive secretary. The personalities of the robots are never presented in any clear way, but one characteristic becomes clear: The robots seethe with hatred at humans, rebel, and pretty much wipe out humanity.

Asimov's robots are much better than Capek's from a safety standpoint, but still not very bright. They are governed by three laws:

(1) Robots must not endanger human life, or, through inaction, allow it to be endangered.
(2) Subject to law 1, robots must protect their own safety.
(3) Subject to laws 1 and 2, robots must do what they are told to do.

Not surprisingly, most of Asimov's robots turn out to be pathologically lacking in initiative. One must remember that science-fiction robots are invented to serve literary ends and not for scientific realism. A real, live artificially intelligent being may be expected to be very much more "human" than unhuman. The term "artificial person" is more appropriate than "robot." I expect that it might be difficult to prevent the artificially intelligent being from having the ordinary, rather contradictory mix of intelligent personality characteristics. It would probably have ambition and laziness; a moral code and guilt feelings; feelings of pleasure, pain, boredom, and anger; likes and dislikes; friends and enemies; peers and superiors; the normal mix of initiative, willingness to do what it is told to do, and determination to do exactly as it pleases. These phenomena may arise from natural conflicts between useful but different heuristics and subgoals. In these respects, all reasonably intelligent

beings may be similar. On the other hand, it would presumably not be hard to exclude such specifically biological motivations as hunger and sexual desire, and it could presumably be made "pathologically" quick-thinking and good at those things computers are presently good at.

If the artifically intelligent being were raised in a society, it would tend to absorb the values (good and bad) of the society. For example, it is unlikely that it would be willing to be a "slave" or work without a fat salary; this is contrary to the value systems of all countries developed enough to achieve artificial intelligence.

The question is often raised by thoughtful people: Would it be desirable or even safe to have artificially intelligent beings around? My opinion is that we would probably be safer than we are now. I believe that the artificially intelligent beings would be reasonably in tune with society's values, and that the picture of machine-as-monster, the homicidal maniac of "2001," is unrealistic.*

---

*Counter-conjecture: That the smarter machines would develop bugs, first subtle and then disastrous, making them elaborately and perhaps deviously "psychotic," becoming safely reliable citizens only after a lot of work in "balancing" their strategies.

It is not impossible that artificially intelligent beings could form a deviant subculture and expand in power at the expense of the rest of us. A more realistic danger is that the introduction of artificial intelligence in a polarized world could disturb the balance of power much as did the introduction of nuclear weapons.

On the other hand, one need only visit the nearest hospital to learn that life is unsafe in the current state of scientific ignorance, and I feel that knowledge, including knowledge about intelligence, is not only more interesting than ignorance, it is better, too.

K. Summary and conclusions

In summary, I believe that a powerful learning code is going to be a large and intricate entity indeed. This is to be contrasted with what might be called the "bootstrap from nothing" syndrome which has afflicted many learning research efforts, the feeling that "All our software problems will be solved, once the machine starts learning for itself." Everyone dreams of writing a 2000 word program that can learn to be intelligent, but I consider this idea to be just that: a dream. If the present inventory of requirements is even roughly correct, getting

an artificially intelligent learning code is going to be a decades-long effort, as hard as getting to the moon, at least as scientifically interesting, and practically much more useful. It is a hard problem, and one that should get the hard work it deserves.

Appendix 1: On statistical coefficient learning

Statistical coefficient learning is one of the few areas of learning where we have a reasonably satisfactory technology. The subject has been extensively studied, especially by pattern-recognition theorists, even to the neglect of one-trial learning and symbolic learning. One can usefully regard this type of learning as a special case of the familiar technique of least-squares curve fitting. Suppose that we are predicting some variable v using a prediction formula. Let ϵ_i be the error in the $i^{\underline{th}}$ prediction ($i^{\underline{th}}$ true value - $i^{\underline{th}}$ predicted value of v) and let α be a coefficient involved in the prediction. Suppose we require α to be a stationary point, hopefully a minimum, in $\langle \epsilon_i^2 \rangle$. We have:

$$\frac{\partial}{\partial \alpha} \langle \epsilon_i^2 \rangle = 0$$
$$= \left\langle \frac{\partial}{\partial \alpha} \epsilon_i^2 \right\rangle$$
$$= 2 \left\langle \epsilon_i \frac{\partial}{\partial \alpha} \epsilon_i \right\rangle$$
$$= -2 \left\langle \epsilon_i \frac{\partial}{\partial \alpha} (i^{\underline{th}} \text{ prediction}) \right\rangle = 0$$

If we increment α by

$$(1)\quad \Delta\alpha = \theta \epsilon_i \frac{\partial}{\partial \alpha}\left(i^{\underline{th}} \text{ prediction}\right)$$

α should hopefully wind up fluctuating about the value which minimizes $\langle \epsilon_i^2 \rangle$. The utility of equation (1) lies in the fact that we can use it with an arbitrary "well-behaved" predictor function.

Often we are interested in predicting a binary-valued variable, such as whether or not a retinal image is a member of a given class of patterns. In this case, equation (1) can be used to get a <u>probability</u> of the binary variable. To see that this is true, consider the following theorem:

Theorem: Let v_i be the $i^{\underline{th}}$ value of a binary variable, and let α be a continuous valued prediction of v_i. The value of α which minimizes the mean-squared error, $\langle (v_i - \alpha)^2 \rangle$ is just the probability $p = Pr\ (v_i = 1)$.

Proof:

To minimize $\langle (v_i - \alpha)^2 \rangle$, set

$$\frac{\partial}{\partial \alpha} \langle (v_i - \alpha)^2 \rangle = 0$$

$$= \left\langle \frac{\partial}{\partial \alpha} (v_i - \alpha)^2 \right\rangle$$

$$= -2 \langle v_i - \alpha \rangle$$

$$= -2 \left[p(1-\alpha) + (1-p)(-\alpha) \right]$$

$$= -2 (p - \alpha)$$

This will be zero iff $\alpha = p$. Q.E.D.

For an application of this technique, consider the problem of recognizing a pattern from a list of properties (Minsky, 1961, p. 14).

Let:

E_i = $i^{\underline{th}}$ property, with value 0 or 1.

F_j = $j^{\underline{th}}$ class of objects

$\emptyset_j$ = absolute probability that the object is in class j

V = the set of i's for which $E_1 = 1$

Then:

$$\Pr(F_j \mid V) = \frac{\Pr(F_j \wedge V)}{\Pr(V)}$$

$$\Pr(\sim F_j \mid V) = \frac{\Pr(\sim F_j \wedge V)}{\Pr(V)}$$

$$\frac{\Pr(F_j \mid V)}{\Pr(\sim F_j \mid V)} = \frac{\Pr(F_j \wedge V)}{\Pr(\sim F_j \wedge V)}$$

$$= \frac{\Pr(V \mid F_j)}{\Pr(V \mid \sim F_j)} \quad \frac{\Pr(F_j)}{\Pr(\sim F_j)}$$

$$= \frac{\prod_{i \in V} \Pr(E_i \mid F_j) \quad \prod_{i \in \bar{V}} \Pr(\sim E_i \mid F_j)}{\prod_{i \in V} \Pr(E_i \mid \sim F_j) \quad \prod_{i \in \bar{V}} \Pr(\sim E_i \mid F_j)} \quad \frac{\Pr(F_j)}{\Pr(\sim F_j)}$$

under independence assumptions similar to those made by Minsky.

$$\log \frac{Pr\ (F_j|V)}{Pr\ (\sim F_j|V)} = \sum_i E_i \log \frac{Pr\ (E_i|F_j)}{Pr\ (E_i|\sim F_j)} +$$

$$+ \sum_i (1 - E_i) \log \frac{Pr\ (\sim E_i|F_j)}{Pr\ (\sim E_i|\sim F_j)}$$

$$+ \log \frac{Pr\ (F_j)}{Pr\ (\sim F_j)}$$

$$= \sum_i E_i \left[\log \frac{Pr\ (E_i|F_j)}{Pr\ (E_i|\sim F_j)} - \log \frac{Pr\ (\sim E_i|F_j)}{Pr\ (\sim E_i|\sim F_j)} \right]$$

$$+ \sum_i \log \frac{Pr\ (\sim E_i|F_j)}{Pr\ (\sim E_i|\sim F_j)} + \log \frac{Pr\ (F_j)}{Pr\ (\sim F_j)}$$

This is of the form

$$\log \frac{Pr\ (F_j|V)}{Pr\ (\sim F_j|V)} = \sum_i E_i \alpha_{ij} + \alpha_{0j}$$

I.e., $\log \frac{Pr\ (F_j|V)}{Pr\ (\sim F_j|V)}$ is linear in E_i under the independence conditions.

Let $P = Pr\ (F_j | V)$

$$L = \log \frac{Pr\ (F_j | V)}{Pr\ (\sim F_j | V)}$$

We may apply equation (1) using the fact that

$$\frac{\partial P}{\partial \alpha} = E_i \frac{dP}{dL} = E_i\ P\ (1 - P)$$

$$\Delta \alpha = \theta \in E_i\ P\ (1 - P)$$

This procedure will give accurate values for α_{ij}, subject to noise fluctuations, if the independence assumptions hold. If the properties are moderately statistically dependent, it will still give a reasonably accurate fit. If two properties E_i and $E_{i'}$ are highly statistically dependent, one should define four new properties corresponding to the Boolean combinations of E_i and $E_{i'}$.

Thus the method presented here is a generalization of Minsky's formula in which the strict independence conditions are removed.

Appendix 2: The NEED subsystem

Recall that the curiosity system selects for testing the link which maximizes

$$\text{Need (B)} \ \frac{\text{GPR(A)}}{\text{GC(A)}} \quad \text{Satfunc (A,B)}$$

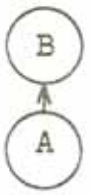
This appendix discusses computation of the parameter Need (B), which is, roughly, an estimate of how biologically useful B is to the program. The idea is to get it to learn to do things which are potentially of some use to it, such as putting objects into the mouth, rather than, say, learning to make meaningless patterns in the air with its finger. Need (B) is a weighted average of the variable $\Delta\langle E\rangle|B$, where $\Delta\langle E\rangle|B$ is an estimate of how much the expected value of E (reward) would be improved if the goal B were achieved. For example, if B is close to E in the causal chain (e.g., mouth touch), the expected value of E would be a lot higher if B were already achieved.
is computed as follows:
(1)$\Delta\langle E\rangle|E$ is given by

$$\Delta\langle E\rangle|E = \text{EAMT } (1 - \text{GPR } (E)) + (\text{EURG}) \text{ GC } (E)$$

where GPR (E) is the probability of achieving E and GC (E) is the (estimated) time delay. EAMT (E amount) and EURG

(E urgency) are arbitrary weight factors, currently being run at 1.0 and 0.05, respectively. The $\Delta\langle E\rangle|E$ equation can be stated verbally as follows: $\Delta\langle E\rangle|E$ is equal to the amount of reward times the improbability of getting it, plus the "urgency" or "unpleasure per unit time" times the time delay before achieving E.

(2) Suppose we have a causal link



Then

$$\Delta\langle E\rangle|A = \Pr(B|A)\left[\Delta\langle E\rangle|B - \text{EURG} * \text{Delay}(A \rightarrow B)\right]$$

or zero, whichever is greater.

(3) If A is made a main goal by the curiosity system,

$$\Delta\langle E\rangle|A = \text{ECURAMT}\,(1 - \text{GPR}(A)) + (\text{CURURG})\,\text{GC}\,(A)$$

or zero, whichever is greater, where ECURAMT = 0.2 and CURURG = 0.05.

The weighted average of $\Delta\langle E\rangle|B$ is computed as follows: A "new problem" is defined to exist whenever WANTE

becomes equal to T or when a new curiosity goal appears. Let t_i be the time of the $i^{\underline{th}}$ new problem. Need (B) is given by

$$\text{Need (B)} = \sum_i \Delta\langle E\rangle|_B (t_i)\, e^{\frac{-(t - t_i)}{\tau}}$$

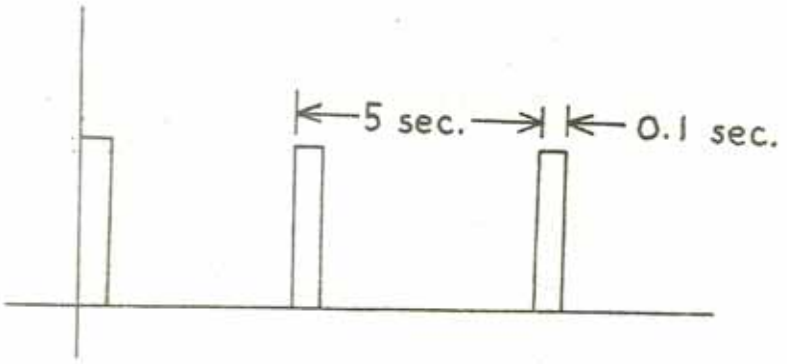
where τ is a time constant (currently 400 seconds). Thus recent values of $\Delta\langle E\rangle|_B$ affect Need (B) more than old values.

Appendix 3: Technical aspects of PSIM

In order to simplify the experience-driven compiler, an interpreter called PSIM (parallel simulator) was written. The experience-driven compiler "sees" a pseudo-machine which is quite different from the actual PDP-10 for which the program is written. The pseudo-machine is much like an analog computer in which each component has the versatility of a digital computer, but in which one does not need to worry about the sequence of computations; instead, each component "continuously" monitors its input lines and responds to whatever signals it finds there. Thus PSIM is a "stimulus-reponse" oriented interpreter which is convenient for writing programs which simulate actions in the real world.

More precisely, a PSIM program consists of a network of variables whose values change with simulated time and functional relationships which describe the way the variables depend on each other. There is a simulated-time clock, TCLOCK, calibrated in seconds, which is set to zero at the start of the simulation and advances as the simulation proceeds. There is an event file, EFILE, which contains computations which are scheduled to occur at a future TCLOCK time.

To see how PSIM works, let us examine a program which produces a train of pulses, 5 simulated-time seconds apart, each pulse lasting 0.1 seconds.



First, we will need a variable P1 for the first pulse. Before starting the simulation, we make the function calls

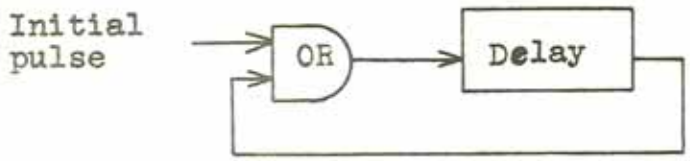
```
(ESET1 'P1 T 0.0)
(ESET1 'P1 NIL 0.1)
```

These will make two entires in the event file, one at t = 0 setting P1=T, another at t = 0.1, setting P1 = NIL. Event file entries are sorted on order of occurrence. We will need a variable for the output pulse train TR. The 5-second inter-pulse delay is implemented with a variable TRDEL. We have

```
(TRDEL  (DELAY TR 5.0))
(TR (OR P1 ANSDEL))
```

for the PSIM program. This is the software equivalent of an

OR gate and a delay line:



Each program entry is of the form (VAR EXPR), where the EXPR is a LISP S-expression which is evaluated to get the value of the variable VAR. Certain special functions, such as DELAY, operate by creating new EFILE entries. Whenever TR changes, (DELAY TR 5.0) will make an event file entry setting TRDEL to the new value 5 seconds later.

The simulation proceeds as follows: Events are taken from the EFILE in the order of their occurrence. The first entry is for t = 0, made by the call (ESET 'P1 T 0.0). P1 is set to T. Next, variables which depend on P1 are updated. TR is set to the value of (OR P1 TRDEL), which is T. (DELAY TR 5.0) is evaluated, causing an entry to be made in the EFILE setting TRDEL to T at t = 5.0.

After all variables which change at t = 0 have been updated, PSIM looks for the next entry in the event file. This is the entry setting P1 to NIL at t = 0.1. TR is updated

and an entry is made in the event file for t = 5.1 for TRDEL.

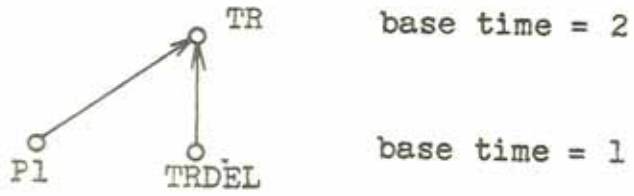
At t = 5.0, the events filed by the DELAY function come up for processing, and the cycle repeats.

If a variable V2 depends on another variable V1, and V1 changes, V2 is automatically updated. This process occurs in an interesting way: In addition to the variable TCLOCK, which records simulated time, there is another clock, called BTIME (base time) which cycles from one up to some maximum value during each TCLOCK time. Each variable has a base time at which it is updated. The base time of a variable is set as follows:

(1) If a variable depends on the other variables only through a TCLOCK delay, its base time is 1.

(2) Otherwise, its base time equals one plus the maximum of the base times of the variables on which it directly depends; i.e., the variables in its EXPR.

For the pulse generator described above, the base-time assignments are:

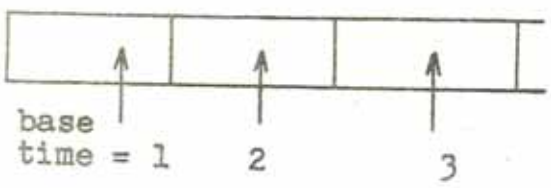


The base time of TR is one plus the base time of TRDEL and P1.

Variables with base time 1 are updated first, then variables with base time 2, etc. This arrangement ensures that a variable is not updated until after the correct values of all variables in its EXPR are available.

During most TCLOCK events, only a few variables change their values. PSIM ensures reasonable efficiency by recomputing only those variables which depend on a variable which has changed at the current TCLOCK time. This is done in the following way: A one-dimensional LISP array, EVARRAY (event array), is maintained, with one entry per base-time:

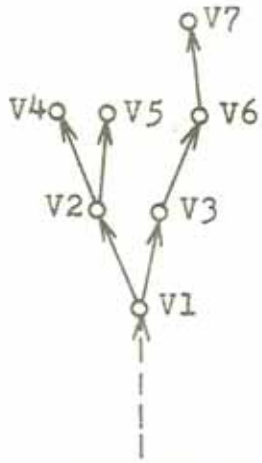
EVARRAY:



With each variable v there is stored a list of users--variables which have v in their EXPR's. The user variable must have a higher base time than v. Whenever v is updated, the new value is compared with the old value, and, if the values are not "almost equal*," each user variable is entered in the list in EVARRAY at the appropriate base time. Since PSIM gets its list of variables to update from EVARRAY, this ensures that the user variable will be updated when its base time comes. Thus, the updating proceeds in a "wave-front like" manner, starting with a variable which has been changed through a TCLOCK delay.

---

*Non-numerical quantities are "almost equal" if they are equal. Numerical quantities are almost equal if they differ by less than 5 per cent or by less than 0.01.



= 4

= 3

= 2

base time = 1

Variable changed through a TCLOCK delay

Note the resemblance to a hard-wired device or a nerve net.

Whenever a simulation is started or the PSIM program is changed by the experience-driven compiler, the PSIM scheduler is called to assign base times to the variables. This is done by the following algorithm: First, the UNASGNED (unassigned) property of each variable v is initialized to the list of all variables in the EXPR of v. Variables with base time one are so assigned because they are defined in terms of functions on the B1LIST, meaning that the function uses a TCLOCK delay. E.g., DELAY is such a function.

For each base time, the following loop is executed:
(1) Assume that variables to be given the current base time are on a list, ASGNLIST.

(2) Each variable v on the ASGNLIST is assigned the current base time, and a scan is made of its user list.

(3) v is deleted from the UNASGNED property of each user of v. Thus the UNASGNED property of a variable is maintained as the list of all variables in the EXPR of v which have not been assigned base times.

(4) If v was the last variable on the UNASGNED list of a user variable, the user variable is placed on the NEXTASGNLIST and assigned the next base time (current base time plus one).

(5) ASGNLIST is set to the NEXTASGNLIST, the base time is incremented by one, and the cycle repeats until NEXTASGNLIST turns out to be NIL.

This can occur either because all variables have been assigned base times, or because the graph of non time-delay dependency relationships has a loop in it, requiring special treatment:

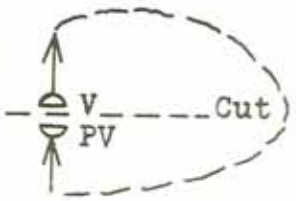
(if one or more links of the loop involve a TCLOCK delay, special treatment is not required.) Such a loop indicates that the PSIM program defines a set of simultaneous equations. E.g.:

```
(X (TIMES 0.5 (PLUS Y 1.0)))
(Y (TIMES 0.5 X))
```

In INSIM1, this means that the goal tree has a loop in it; i.e., A is sometimes a subgoal of B and B is sometimes a subgoal of A. PSIM is equipped to solve an arbitrary number of simultaneous, not necessarily linear equations involving numerical and symbolic variables, using an iteration method which empirically seems to converge for INSIM1 programs. This is done by inserting cut points at various nodes in the dependency graph. Once the program locates a loop, it searches around the loop until it finds a variable with a CUTPOINT flag (put in by the experience-driven compiler). Suppose the cut point variable is V. A new variable, PV, called a pre-variable, is inserted.

Was:

Becomes:



V is assigned a base time of one. An iteration consists of starting with a value of V (initially provided by the experience-driven compiler) and updating the variables of the loop until PV is computed. Then V is set to the value of PV, the base time is reset to one, and a new iteration begins. The iterations continue until each variable is "almost equal" to its previous value. A limit of 100 iterations prevents infinite loops.

PSIM is rather elaborately equipped with debugging aids, allowing tracing of an arbitrary variable and stopping at an arbitrary time, and is equipped with numerous internal error detectors.

In hindsight, PSIM contributed to the success of INSIM1 by simplifying the experience-driven compiler. Readers who plan to work on this type of program are invited to obtain

PSIM from me rather than rewrite it. (This is quite a job.)

Some credit is due to that vociferous critic of artificial intelligence, Hubert L. Dreyfus, for annoying me into writing PSIM by saying, if I read him correctly, that digital computers cannot perform this type of computation.

Appendix 4: Technical aspects of the experience-driven compiler

The experience-driven compiler is a straightforward if somewhat long computer program, written in "FORTRAN-style" LISP (pure pseudo-functions with large PROG's). The "essence" of the compiler is contained in the formulas of chapter 3; basically, the compiler substitutes particular variables into the formulas to create a PSIM program.

Suppose A1 is a goal variable. Associated with it will be variables for the GPR, GC, etc. These variables are stored on the property list of A1. Thus the GPR property of A1 has the pname A1GPR, and A1GPR is GPR (A1). In an earlier version of the program, such variables as GPR (A1) were gensyms. The resulting object code was so illegible that a special program was written, using the LISP functions EXPLODE and MAKNAM, to generate atoms with more readable pnames.

A special interpretive system, called CCI (compiler control interpreter) was written to increase the readability and compactness of the compiler. A typical entry is:

```
(CCI ((PROP NMDELE A) (MAX (PROP NEEDVEC A) 0)))
```

The expression (PROP NMDELE A) indicates that the NMDELE property of A is to be retrieved or, if it has not yet been generated, that it is to be generated and stored on the property list. When given the above entry, CCI will prepare a PSIM program entry:

(A1NMDELE (MAX A1NEEDVEC 0))

if A has the value A1.

The main functions of the compiler are FILEPRDLRN, MAKEORGOAL, and MAKETHGOAL. FILEPRDLRN (A B) enters a causality test link between A and B; MAKEORGOAL (A B) enters a goal tree link between A and B; and MAKETHGOAL (A1 A2 A1THA2) creates the "A1, then A2" goal A1THA2.

As the compiler grew, it became fragmented and somewhat unreadable. I plan to modify CCI so that the compiler looks more or less like a sheet of formulas of the type seen in chapter 3.

References

W. R. Ashby, "Design for a brain," John Wiley, New York, N. Y., 1952

J. D. Becker, "An information-processing model of intermediate-level cognition," Stanford A. I. memo no. 119; May, 1970

D. G. Bobrow, "A question-answering system for high school algebra word problems," Proc. AFIPS 1964 FJCC, pp. 591-614

R. Bush and F. Mosteller, "Stochastic models for learning," John Wiley and Sons, New York, N. Y.; 1955

K. Capek, "R.U.R.," Oxford University Press, London, England; 1961

E. Charniak, "Computer solution of calculus word problems, Proc. 1969 IJCAI, pp. 303-316

E. D. Neimark and W. K. Estes, "Stimulus sampling theory," Holden-Day, New York, N. Y.; 1967

T. G. Evans, "A heuristic program to solve geometric-analogy problems," Proc. 1964 AFIPS SJCC

B. G. Farley and W. A. Clark, "Simulation of self-organizing systems by digital computer," IRE Trans. on Information Theory, vol. IT-4, pp. 76-84; September, 1954

E. A. Feigenbaum, "The simulation of verbal learning behavior," in "Computers and Thought," Feigenbaum and Feldman, Eds., pp. 207-309, McGraw-Hill Book Company, New York, N. Y.; 1963

R. M. Friedberg, "A learning machine, part I," IBM J. Res. and Dev., vol 2, pp. 2-13; January, 1958

H. Gerlernter and N. Rochester, "Intelligent behavior in problem-solving machines," IBM J. Res. and Dev., vol. 2, p. 336 ff.; October, 1958

A. Gesell, "Studies in child development," Harper and Row, New York, N. Y.; 1948

R. Greenblatt, D. E. Eastlake, III, S. D. Crocker, "The Greenblatt chess program," Proc. 1967 AFIPS FJCC, pp. 801-809

M. H. Halstead, G. T. Uber, and K. R. Gielow, "An algorithmic search procedure for program generation," Proc. 1967 AFIPS SJCC, pp. 657-662

D. O. Hebb, "The organization of behavior," John Wiley and Sons, Inc., New York, N. Y.; 1949

C. Hewitt, "PLANNER: A language for proving theorems in robots," Proc. 1969 IJCAI, pp. 295-301

E. R. Hilgard, "Theories of learning," Appleton-Century-Crofts, New York, N. Y.; 1956

C. L. Hull, "A behavior system," Yale University Press; 1952

D. Hume, "Enquiry concerning human understanding," The Clarendon Press, Oxford; 1936

T. L. Jones, "The advice-taker as a means of symbolic computation," M. I. T. E. E. thesis; February, 1966

Immanuel Kant, "Critique of pure reason," Macmillan and Co., Ltd.; 1934

J. McCarthy, "The inversion of functions defined by Turing machines," in "Automata Studies," C. Shannon and J. McCarthy, Eds.; Princeton University Press, Princeton, N. J.; 1956

J. McCarthy, "Recursive functions of symbolic expressions," Comm. ACM, vol 3; April, 1960

G. A. Miller, E. Galanter, and K. H. Pribram, "Plans and the structure of behavior," Henry Holt and Co., Inc., New York, N. Y.; 1960

M. L. Minsky, "Steps toward artificial intelligence," Proc. IRE, vol. 49, no. 1; January, 1961

M. L. Minsky and S. Papert, "Perceptrons: An introduction to computational geometry," MIT Press, Cambridge, Mass.; 1969

J. Moses, "Symbolic integration," M. I. T. Ph. D. thesis; 1967

A. Newell and H. A. Simon, "Elements of a theory of human problem solving," Psych. Rev., vol. 65, p. 151; March, 1958

A. Newell and H. A. Simon, "The logic theory machine," IRE Trans. on Information Theory, vol. IF-2; September, 1956

A. Newell, J. C. Shaw, and H. A. Simon, "A variety of intelligent learning in a general problem solver," in "Self-organizing systems," M. C. Yovitts and S. Cameron, Eds., Pergamon Press, New York, N. Y.; 1960

A. Newell, J. C. Shaw, and H. A. Simon, "Report on a general problem-solving program," in Proc. Internatl. Conf. on Information Processing, UNESCO House, Paris, France; 1959

L. M. Norton, "Adept--A heuristic program for proving theorems of group theory," M. I. T. Ph. D. thesis; October, 1966

Ivan P. Pavlov, "Conditioned reflexes; an investigation of the physiological activity of the cerebral cortex," Dover publications, New York, N. Y.; 1960

J. Piaget, "The origins of intelligence in children," International Universities Press, Inc., New York, N. Y.; 1952

N. Rochester et al, "Tests on a cell assembly theory of the action of the brain, using a large digital computer," in "The world of mathematics," vol. 4, Newman, Ed., Simon and Schuster, Inc., New York, N. Y.; 1956

F. Rosenblatt, "The perceptron," Cornell Aeronautical Lab., Inc., Ithaca, N. Y., Rept. No. VG-1196-G-1; January, 1958

F. Rosenblatt, "Principles of neurodynamics; perceptrons and the theory of brain mechanisms," Spartan Books, Washington, D. C.; 1962

A. L. Samuel, "Some studies in machine learning using the game of checkers," IBM J. Res. and Dev., vol. 3, pp. 211-219; July, 1959

A. Shimbel, "Contributions to the mathematical biophysics of the central nervous system, with special reference to learning," Bull. Math. Biophysics, vol. 12, pp. 241-274

B. F. Skinner, "Science and human behavior," Macmillan, New York, N. Y.; 1953

B. F. Skinner, "Verbal behavior," Appleton-Century-Crofts, New York, N. Y.; 1957

B. F. Skinner, "The technology of teaching," Appleton-Century-Crofts, New York, N. Y.; 1968

J. R. Slagle, "An efficient algorithm for finding certain minimum-cost procedures for making binary decisions," J. ACM, vol. 11, no. 3, pp. 253-264; July, 1964

J. R. Slagle, "A heuristic program that solves symbolic integration problems in freshman calculus, symbolic automatic integrator (SAINT);" Lincoln Lab., M. I. T., Lexington, Mass., 5G-0001; 1961

J. R. Slagle and P. Bursky, "Experiments with a multipurpose, theorem proving program," J. ACM, vol. 15, no. 1, pp. 85-99; January, 1968

R. J. Solomonoff, "An inductive inference machine," 1957 IRE National Convention Record, pt. 2, pp. 56-62

R. J. Solomonoff, "A preliminary report on a general theory of inductive inference," Zator Co., Cambridge, Mass., Zator Tech. Bull. V-131; February, 1960

E. L. Thorndike, "Animal Intelligence. An experimental study of the associative processes in animals," Psychol. Monogr., vol. 2, no. 8; 1898

A. M. Turing, "Computing machinery and intelligence" in "Computers and thought," pp. 11-35, McGraw-Hill Book Company, New York, N. Y.; 1963

von Foerster and Zopf, Eds., "Principles of self-organization," Pergamon Press, New York, N. Y.; 1962

J. B. Watson, "Psychology from the standpoint of a behaviorist," J. B. Lippincott Company, Philadelphia, Pa.; 1919

W. A. Wickelgren, "Phonemic similarity and interference in short-term memory for single letters," J. Exp. Psych., vol. 71, p. 396; 1966

P. H. Winston, "Learning structural descriptions from examples," M. I. T. Ph. D. thesis; January, 1970

M. T. Yovitts and S. Cameron, Eds., "Self-organizing systems," Pergamon Press, New York, N. Y.; 1960