CMU-CS-83-141

A UNIVERSAL WEAK METHOD

John Laird and Allen Newell
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213
June 1983

ABSTRACT

The weak methods occur pervasively in AI systems and may form the basic methods for all intelligent systems. The purpose of this paper is to characterize the weak methods and to explain how and why they arise in intelligent systems. We propose an organization, called a *universal weak method*, that provides functionality of all the weak methods. A universal weak method is an organizational scheme for knowledge that produces the appropriate search behavior given the available task-domain knowledge. We present a problem solving architecture, called SOAR, in which we realize a universal weak method. We then demonstrate the universal weak method with a variety of weak methods on a set of tasks.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Table of Contents

List of Figures

A UNIVERSAL WEAK METHOD[1]

A basic paradigm in artificial intelligence (AI) is to structure systems in terms of *goals* and *methods*, where a goal represents the intention to attain some object or state of affairs, and a method specifies the behavior to attain the desired objects or states. Some methods, such as hill climbing and means-ends analysis, occur pervasively in existing AI systems. Such methods have been called *weak methods*. It has been hypothesized that they form the basic methods for all intelligent systems (Newell, 1969). Further, it has been hypothesized that they all are methods of searching problem spaces (Newell & Simon, 1972; Newell, 1980a).

Whatever the ultimate fate of these hypotheses, it is important to characterize the nature of weak methods. That is the purpose of this paper. We propose that the characterization does not lead, as expected organization, with a collection of the weak methods plus a method-selection mechanism. Instead, we propose a single organization, called a *universal weak method*, embedded within a *problem-solving architecture*, that responds to a situation by behaving according to the weak method appropriate for the agent's knowledge of the task.

Section 1 introduces weak methods as specifications of behavior. Section 2 introduces search in problem spaces and relates it to weak methods. The concepts in these first two sections are familiar, but it is useful to provide a coherent treatment as a foundation for the rest of the paper. Section 3 introduces a specific problem-solving architecture, based on problem spaces and implemented in a production system, that provides an appropriate organization within which to realize weak methods. Section 4 defines a universal weak method and its realization within the problem-solving architecture. Section 5 demonstrates an experimental version of the architecture and the universal weak method. Section 6 discusses the theory and relates it to other work in the field. Section 7 concludes.

---

[1] A brief report of the results of this paper was presented at IJCAI-83 (Laird & Newell, 1983).

1. The Weak Methods

We start by positing an *agent* with certain capabilities and structure. We are concerned ultimately with the behavior of this agent in some environment and the extent to which this behavior is intelligent. We introduce goals and methods as ways of specifying the behavior of the agent. We then concentrate on methods that can be used when the agent has little knowledge of its environment, namely the *weak methods*.

1.1. Behavior specification

Let the agent be characterized by being, at any moment, in contact with a *task environment* (or *task domain*), which is in some state, out of a set of possible states. The agent possesses a set of *operators* that affect the state of the environment. The *behavior* of the agent during some period of time is the sequence of operators that occur during that period, which induces a corresponding sequence of states of the environment. For the purposes of this paper, certain complexities can be left to one side: concurrent or asynchronous operators, continuously acting operators, and autonomous behavior by the environment.

The structure of the agent can be decomposed into two parts, *[C, Q]*, where Q is the set of operators and C is the *control*, the mechanism that determines which operator of Q will occur at each moment, depending on the current environmental state and the past history. Additional structure is required for a general intelligent agent, namely, that it be a *symbol system* (Newell, 1980a), capable of universal computations on symbolic representations. This permits, first of all, the creation of internal representations that can be processed to control the selection of external operators. These internal representations can also be cast as being in a state, out of a set of possible states, with operators that change states internally. It is normal in computer science not to distinguish sharply between behavior in an external task environment and in an internal task environment, since they all pose the same problems of control.

Having a symbol system also permits the control to be further decomposed into $C = [I, S]$, where S is a symbolic *specification* of the behavior to be produced in the task environment and I is an *interpreter* that produces the behavior from S in conjunction with the operators and the state. It is natural to take S to be a *program* for the behavior of the agent. However, neither the form nor the content of S is given. In particular, it should not be presumed to be limited to the constructs available from familiar programming languages: sequences, conditionals, procedures, iterations and recursions, along with various naming and abstraction mechanisms. Indeed, a basic scientific problem for AI is to discover how future behavior of an agent is represented symbolically in that agent, so as to produce intelligent action.

A critical construct for specifying behavior is a *goal*. A goal is a symbolic expression that designates an object or state of affairs that is to be attained. Like other control constructs it serves to guide behavior when it occurs in a specification, S, and is properly so interpreted by I. Goal objects or situations can be specified by

means of whatever descriptive mechanisms are available to the agent, and such descriptions may designate a unique situation or a class of situations. A goal does not state what behavior is to be used to attain the goal situation. This part of the specification is factored out and provided by other processes in the agent which need not be known when the goal is created. However, there must exist a *selection* process to determine what behaviors will occur to attain the goal. The goal does not include the details of this selection process; however, the goal may contain auxiliary information to aid the selection, such as the history of attempts to attain the goal.

An *AI method* (hereafter, just a *method*) is simply a specification of behavior that includes goals along with all the standard programming-language constructs. The creation of a subgoal, as dictated by the method, is often divorced from the attempt to attain the subgoal. This separation gives rise to the familiar *goal hierarchy*, which consists of the lattice of subgoals and supergoals, and the *agenda mechanism*, which keeps track of which subgoals to attempt. Goals, methods, and selection processes that link goals to methods provide the standard repertoire for constructing current AI systems.[2]

1.2. The definition of a weak method

Methods, being an enhancement of programs, can be used for behavioral specifications of all kinds. Indeed, a major objective of the high-level AI languages was to make it easy to create methods that depended intricately upon knowledge of the specific task domain. We consider here the other extreme, namely, methods that are extremely general:

> A *weak method* is an AI method (a specification of behavior using goals and the control constructs of programming languages) with the following two additional features:
>
> 1. It makes limited demands for knowledge of the task environment.
>
> 2. It provides a schema within which domain knowledge can be used.

Any method (indeed, any behavior specification) makes certain demands on the nature of the task environment for the method to be carried out. These can be called the *operational demands*. Take for example the goal to find the deepest point of a lake. One method is to use a heavy weight with a rope, and keep moving the weight as long as the anchor goes deeper. One operational demand of this method is that the task environment must include a weight and a rope, and their availability must be known to the agent. Additional *effectiveness demands* also can exist. Even if the method can be performed (the anchor and rope are there and can be moved), the method may attain the goal only if other facts hold about the environment

---

[2] Explicit goal structures have been with AI almost from the beginning (Feigenbaum & Feldman, 1963); but did not become fully integrated with programming language constructs until the high-level AI languages of the early 1970s (Hewitt, 1971; Rulifson, Derksen & Waldinger, 1972). Because of the flexibility of modern Lisp systems, the practice remains to construct ad hoc goal systems, rather than use an integrated goal-containing programming language (however see Kowalski, 1979).

(the lake does not have different deep areas with shallows between them).

The fewer the operational demands, the wider the range of environments to which a method can be applied.[3] Thus, highly general methods make weak demands on the task environment, and they derive their name from this feature. Presumably, however, the less that is specified about the task environment, the less effective the method will be. Thus, in general, methods that make weak demands provide correspondingly weak performance, although the relationship is not invariable.

The intrinsic character of problematical situations (situations requiring intelligence) is that the agent does not know the aspects of the environment that are relevant to attaining the goal. Weak methods are exactly those that are useful in such situations, because not much need be known about the environment to use them. Weak methods occur under all conditions of impoverished knowledge. Even if a strong method is ultimately used, an initial method is required to gather the information about the environment required by the strong method. This initial method must use little knowledge of the environment, and therefore is a weak method. Thus the major question about weak methods is not so much whether they exist, but their nature and variety.

Weak methods can use highly specific knowledge of the domain, despite their being highly general. They do this by requiring the domain knowledge to be used only for specific functions. For instance, if a weak method uses an evaluation function, then this evaluation function can involve domain-specific knowledge, and an indefinitely large amount of such knowledge. But the role of that knowledge is strongly proscribed. The evaluation function is used only to compare states and select the best. A weak method is in effect a method schema that admits many instantiations, depending on the domain knowledge that is used for the specific functions of the method.

1.3. The common weak methods

The two defining features of weak methods do not delineate sharply a subclass of methods. Rather, they describe the character of useful methods that seem in fact to occur in both artificial-intelligence systems and human problem solving. Figure 1-1 provides a list of common weak methods, giving for each a brief informal definition. Consider the method of hill climbing (HC). It posits that the system is located at a point in some space and has a set of operators that permit it to move to new points in the space. It also posits the existence of an evaluation function. The goal is to find a point in the space that is the maximum of this evaluation function. The method itself consists of applying operators to the current point in the space to produce adjacent points, finally selecting one point that is higher on the evaluation function and making it the new

---

[3] There is more to it than this, e.g., whether the knowledge demanded by the method is likely to be available to a problem solver who does not understand the environment.

current point. This behavior is repeated until there is no point that is higher, and this final point is taken to attain the goal.

Generate and test (GT). Generate candidate solutions and test each one; terminate when found.

Hill climbing (HC). To find a maximum point in a space, consider the next moves (operators) from a given position and select one that increases the value.

Simple hill climbing (SHC). Hill climbing with selection of the first operator that advances.

Steepest ascent hill climbing (SAHC). Hill climbing with selection of the operator that makes the largest advance.

Heuristic search (HS). To find an object in a space generated by operators, move from the current position by considering the possible operations and selecting an untried one to apply; test new positions to determine if they are a solution. In any event, save them if they could plausibly lead toward a solution; and choose (from the positions that have been saved) likely positions from which to continue the search.

Means-ends analysis (MEA). In a heuristic search, select the next operator to reduce the difference between the current position and the desired positions (so far as they are known).

Depth-first search (DFS). To find an object in a space generated by operators, do a heuristic search but always move from the deepest position that still has some untried operators.

Breadth-first search (BrFS). To find an object in a space generated by operators, do a heuristic search, but always move from the least-deep position that still has some untried operators.

Best-first search (BFS). To find an object in a space generated by operators, do a heuristic search, but always move from a position that seems most likely to lead to a solution and still has some untried operators.

Modified best-first search (MBFS). To find an object in a space generated by operators, do a best-first search, but once a position is selected from which to advance, try all the operators at that position before selecting a new position.

A*. Modified best-first search when it is desired to find the goal state at the minimum depth from the starting position.

Operator subgoaling (OSG). If an operator cannot be applied to a position, then set up a subgoal of finding a position (starting from the current position) at which the operator can be applied.

Match (MCH). Given a form to be modified and completed so as to be identical to a desired object, then compare corresponding parts of the form and object and modify the mismatching parts to make them equal (if possible).

Figure 1-1: Some common weak methods.

The operational demands of this method are dictated directly from the prescribed computations. It must be possible to: (1) represent a state in the space; (2) apply an operator to produce a new state; and (3) compare two adjacent states to determine which is higher on the evaluation function.[4] These demands are all expressed in terms of the agent's abilities; thus, the demands on the task environment are stated only indirectly. It is a

[4] The method requires additional capabilities of the agent that do not involve the environment, such as selecting operators.

matter of analysis to determine exactly what demands are being made. Thus, hill climbing normally occurs with an explicitly given evaluation function, so that the comparison is done by evaluating each state and comparing the results. But all that is required is comparison between two states, not that an explicit value be obtainable for each state in isolation. Furthermore, this comparison need not be possible on all pairs of states, but only on adjacent states, where one arises from the other by operator application. Each of these considerations weakens the demands on the environment, while still permitting the method to be applied.

For hill climbing, the difference between operational demands and effectiveness demands is a familiar one. The state with the absolute highest value will always be reached only if the space is unimodal; otherwise only a local hill is climbed, which will yield the global optimum only if the method starts on the right hill. There also exist demands on the environment for the efficiency of hill climbing which go beyond the question of sheer success of method, such as lack of plateaus and ridges in the space.

Hill climbing is defined in terms of general features of the task environment, namely the operators and the evaluation function. These reflect the domain structure. The method remains hill climbing, even if arbitrarily large amounts of specific domain knowledge are embodied in the evaluation function. But such knowledge only enters the method in a specific way. Even if the knowledge used in the evaluation function implies a way to go directly to the top of the hill, there is no way for such an inference to be detected and exploited.

Many variations of a weak method can exist. For example, hill climbing leaves open exactly which operators will be applied and in which order (if the system is serial); likewise, it does not specify which of the resulting points will be chosen, except that it must be higher on the hill. In *simple hill climbing* (SHC) the operators are generated and applied in an arbitrary order, with the first up-hill step being taken; in *steepest ascent hill climbing* (SAHC), all the adjacent points are examined and the one that is highest is taken. Under different environments one or the other of these will prove more efficient, although generally they will both ultimately climb the same hill.

Subgoals enter into hill climbing only if the acts of selecting an operator, applying it or evaluating the result are problematical and cannot be specified in a more definite way. More illustrative of the role of subgoaling is the method of *operator subgoaling*, which deliberately sets up the subgoal of finding a state in which the given operator is applicable. This subgoal is to be solved by whatever total means are available to the agent, which may include the creation of other subgoals. Operator subgoaling is only one possibility for setting up subgoals, and perhaps it should not be distinguished as a method all by itself. However, in many AI programs, operator subgoaling is the only form of subgoaling that occurs (Fikes, Hart & Nilsson, 1972; Sacerdoti, 1977).

1.4. Complex programs are composed of weak methods

The weak methods in Figure 1-1 occur with great frequency in practice. Many others are known, both for AI systems (e.g., *iterative deepening* in game playing programs), and for humans (e.g., the main method of Polya, 1945). The weak methods appear to be a mainstay of AI systems (Newell, 1969). That is, AI systems rely on weak methods for their problem-solving power. Much behavior of these systems, of course, is specified in highly constrained ways, where the program exhibits limited and stereotyped behavior. The primary exceptions to this occur in modern AI expert systems (Duda & Shortliffe, 1983), which rely as much as possible on large amounts of encoded knowledge to avoid search. But even here many of them are built on top of search methods, e.g,. MYCIN.

To provide an indication of how an AI system can be viewed as a composition of weak methods, the GPS of Newell & Simon (1963) can be described as *means-ends analysis* plus *operator subgoaling*. To carry out these two weak methods, others are required: *matching* is used to compare the current state to the desired one; and *generate and test* is used to select an operator when a difference determines a subset of operators, rather than a unique one. Likewise, *generate and test* is used to select goals to try if a path fails (although this mechanism is not usually taken to be part of the core of GPS). Given these weak methods, little additional specification is required for GPS: the representations and their associated data operations, the mechanisms for constructing and maintaining a goal tree, a table of fixed associations between differences and operators, and a fixed ordering function on the differences. A similar story could be told for many AI programs, such as Dendral, AM, Strips, EL, and others that are less well known (for examples of earlier programs, see Newell, 1969).

To substantiate the claim of the ubiquitous use of weak methods in AI would require recasting a substantial sample of AI programs explicitly in terms of compositions of weak methods and no such attempt has yet been made. Furthermore, no formulation of weak methods yet exists that provides a notion of a basis or of completeness. In any event, this paper does not depend on such issues, only on the general fact that weak methods play an important enough role to warrant their investigation.

2. The Problem Space Hypothesis

Behavior specifications of an agent lead to behavior by being interpreted by an architecture. Many architectures are possible. However, existing families of architectures are designed primarily to meet the requirements of current programming languages, which are behavior specifications that do not include goals or methods.

A key idea on which to base the architecture for an intelligent agent is *search.* AI has come to understand that all intelligent action involves search (Newell & Simon, 1976). All existing AI programs that work on problems of appreciable intellectual difficulty incorporate combinatorial search in an essential way, as the examination of any AI textbook will reveal (Nilsson, 1971; Winston, 1977; Nilsson, 1980). Likewise, human problem-solving behavior seems always to exhibit search (Newell & Simon, 1972), though many forms of difficult and creative intellectual activity remain to be investigated from this viewpoint. On the other hand, the essential role of search in simple, routine or skilled behavior is more conjectural. In itself, the symbolic specification of behavior does not necessarily imply any notion of search, as typical current programming languages bear witness. They imply only the creation at one moment of time of a partial specification, to be further specified at later times until ultimately, at performance time, actual behavior is determined. Nevertheless, the case has been argued that a framework of search is involved in *all* human goal-directed behavior, a hypothesis that is called the *Problem Space Hypothesis* (Newell, 1980b).

We will adopt this hypothesis of the centrality of search and will build an architecture for the weak methods around it. The essential situation is presented in Figure 2-1, which shows abstractly the structure of a general intelligent agent working on a task. As the figure shows, there are two kinds of search involved, *problem search* and *knowledge search.*

2.1. Problem search

Problem search occurs in the attempt to attain a goal (or solve a problem). The current state of knowledge of the task situation exists in the agent in some representation, which will be called a *problem state* (or just a *state.*) The agent is at some initial state when a new goal is encountered. The agent must find a way of moving from its current state to a goal state. The agent has ways of transforming this representation to yield a new representation that has more (or at least different) knowledge about the situation; these transformations will be called *operators.* The set of possible states plus the operators that permit the movement of the agent from state to state will be called the *problem space.* This is similar to the situation described in Section 1.1, except that here the state is a representation that is internal to the agent.

The desired state of the goal can be specified in many ways: as a complete state, an explicit list of states, a pattern, a maximizer of a function, or a set of constraints, including in the latter, constraints on the path

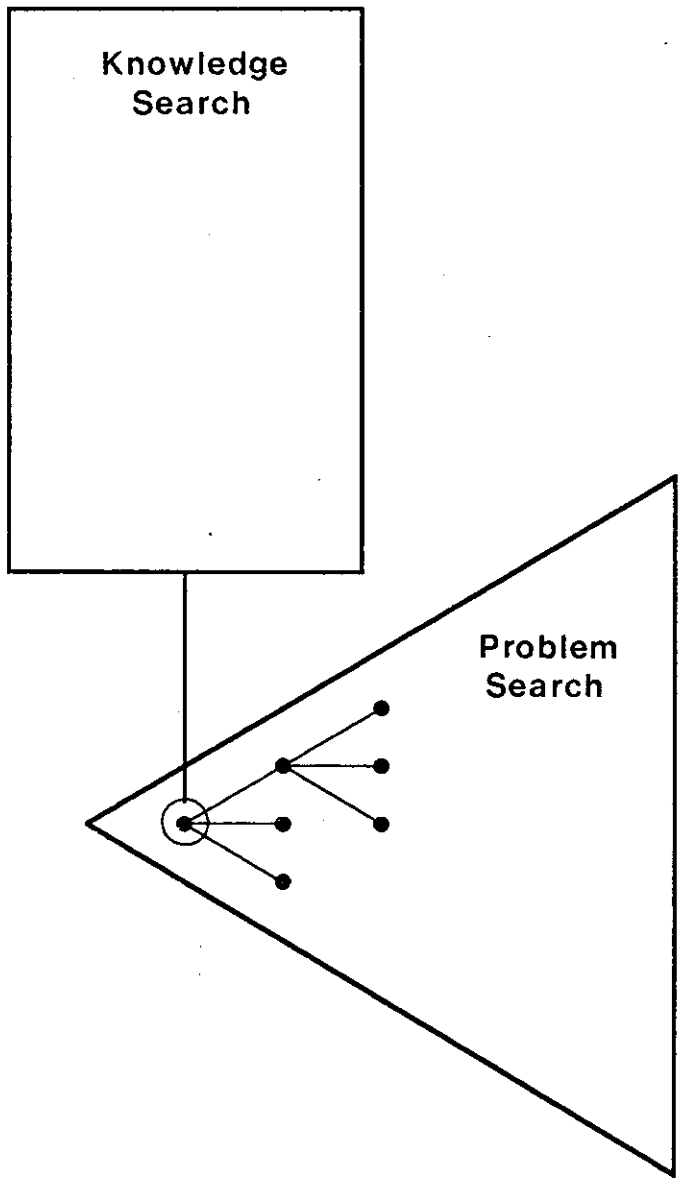

Figure 2-1: The framework for intelligent behavior as search.

followed. The agent must apply a sequence of operators, starting at the initial state, to reach a state that satisfies whatever goal specification is given.

Search of the problem space is necessarily combinatorial in general. To see this, note that the space does not exist within the problem solver as a data structure, but instead is generated state by state by means of operator applications, until a desired state is found. If, at a state, there is any uncertainty about which operator is the appropriate one to apply (either to advance along a solution path or to recover from a nonsolution path), this must ultimately translate into actual errors in selecting operators. Uncertainty at

successive states cascades the errors of operator selection and thus produces the familiar combinatorially branching search tree.[5] Uncertainty over what to do is the essence of the problematic situation. It is guaranteed by the de novo generation of the space, which implies that new states cannot be completely known about in advance.[6]

The uncertainty at a state can be diminished by the agent's knowledge about the problem space and the goal. The task for the agent at each state is to bring this *search-control* knowledge to bear on the functions required at a node of the search tree in Figure 2-1. There is a fixed set of such functions to be performed in searching a problem space (Newell, 1980b):

1. Decide on success (the state is a desired state).
2. Decide on failure or suspension (the goal will not be achieved on this attempt).
3. Select a state from those directly available (if the current state is to be abandoned).
4. Select an operator to apply to the state.
5. Apply the operator to obtain the new state.
6. Decide to save the new state for future use.

In addition there are decisions to be made that determine the goals to be attempted and the problem spaces within which to attempt them. The architecture brings search-control knowledge to bear to perform these functions intelligently. Depending on how much knowledge the agent has and how effective its employment is, the search in the problem space will be narrow and focused, or broad and random.

2.2. Knowledge search

Some process must select the search-control knowledge to be used to make the decisions in the problem space. In a problem solver constructed for a specific and limited task, there is little difficulty in associating the correspondingly limited search-control knowledge with the site of its application. However, a general intelligent agent attempts many tasks and therefore has a large body of potentially applicable knowledge. There is then the need to determine what knowledge is available that is applicable to controlling the search of a particular task. Knowledge is encoded in memory, hence in some extended data base. That is, extended knowledge implies extended memory. Since the problem, as represented in the problem space, is new, the agent cannot know in advance what knowledge is relevant to controlling the search for a solution.

---

[5] In task environments that are densely enough interconnected, actual return to prior states can be avoided in favor of always moving forward from the current "bad" state, seeking a better state; but the essential combinatorial explosion remains.

[6] This argument applies equally well to serial and concurrent processing, as long as there is an overall resource limit, i.e., as long as exponential parallel processing is not possible.

Necessarily, then, the data base must be searched for the relevant knowledge. Figure 2-1 shows search-control knowledge being applied only to a single node in the problem space, but knowledge search must occur at each node in the problem search. Hence, knowledge search lies in the inner loop of problem search and its efficiency is of critical importance.

Knowledge search differs from problem search in at least one important respect. The data base to be searched is available in advance; hence its accessing structure may be designed to make the retrieval of knowledge as efficient as possible. In consequence, the search is not necessarily combinatorial, as is the search in the problem space. The architectural possibilities for the memory that holds the search-control knowledge are not yet well understood. Much work in AI, from semantic nets to frame systems to production systems, is in effect the exploration of designs for search-control memory.

2.3. Goals and methods in terms of search

The mapping of goals and methods into the search structure of Figure 2-1 is easy to outline. A goal leads to forming (or selecting): (1) a problem space; (2) within that space, the initial state, which corresponds to the agent's current situation; and (3) the desired states, which correspond to the desired situation of the goal. A method corresponds to a body of search-control knowledge that guides the selection and application of operators. A method with subgoals (such as operator subgoaling in Figure 1-1) leads to creating a new goal (with a new problem space and goal states), and to achieving it in the subspace. The new problem space need not actually be different from the original; e.g., in operator subgoaling it is not. Upon completion of the subgoal attempt, activity may return to the original space with knowledge from the subgoal attempt. A goal need not be solved within a single problem space, but may involve many problem spaces, with the corresponding goal hierarchy.

The commitment to use search as the basis for the architecture implies that the weak methods must all be encoded as search in a problem space. A glance at Figure 1-1 shows that many of them fit such a requirement -- heuristic search, hill climbing, means-ends analysis, best first search, etc. This hardly demonstrates that all weak methods can be naturally so cast, but it does provide encouragement for adopting a search-based architecture.

3. A Problem Solving Architecture

In this section we give a particular architecture for problem solving, based on the search paradigm of the prior section. We will call it *SOAR*, for *State, Operator and Result*, which represents the main problem-solving act of applying an operator to a state and producing a result. Such an architecture consists of a processing structure where the functions at a node of the search tree are decomposed into a discrete set of actions executable by a machine with appropriate memories and communication links.

3.1. The object context

SOAR has representations for the *objects* involved in the search process: goals, problem spaces, states and operators. Each primitive representation of an object can be *augmented* with additional information. The augmentations are an open set of unordered information, being either information about the object, an association with another object, or the problem-solving history of the object. For example, states may be augmented with an evaluation, goals may be augmented with a set of constraints a desired state must satisfy, problem spaces may be augmented with their operators, and a state may be augmented with information detailing the state and the operator that created it.

As shown in Figure 3-1, the architecture consists of the *current context* and the *stock*. The current context consists of a single object of each type. These objects determine the behavior of the agent in the problem space. The goal in the current context is the current goal; the problem space is the current problem space, and so on. The stock is an unordered memory of all the available objects of each type. The objects in the current context are also part of the stock.

	Goal	Problem Space	State	Operator
Current Context:	$goal_{10}$	$problem\ space_{3}$	$state_{107}$	$operator_{6}$
Stock:	$goal_{102}$	$problem\text{-}space_{14}$	$state_{34}$	$operator_{74}$
	$goal_{30}$	$problem\text{-}space_{23}$	$state_{702}$	$operator_{56}$
	$goal_{231}$	$problem\text{-}space_{1}$	$state_{19}$	$operator_{202}$
	...	...	...	...

Figure 3-1: Stock and current context of SOAR, the problem solving architecture.

The stock is a memory and its structure is necessarily limited by the physical resources of the agent. However, the architecture assumes an *unlimited memory structure*, which implies unlimited access to the

stock, unlimited capacity of the stock and unlimited reliability of the stock. It implies that the subobjects of the objects in the stock (such as the operators of a problem space) are also accessible and that new objects to the system (such as from operator applications or the external environment) become automatically available in the stock. This is a simplifying assumption for purposes of analysis and must ultimately be removed.

A single generic *control act* is available in the architecture: *replacement* of an object in the current context by another object of the same type, from the stock. All of the functions at a node in the search tree are realized by replacements of one kind or another. When activity for a goal ceases, because of success, failure or suspension, a prior goal from the stock replaces the current one. Returning to a prior state is accomplished by a state in the stock replacing the current one. An operator is selected by placing it in the current context. A step in the problem space occurs when the current operator is applied to the current state to produce a new state. This new state is deposited in the stock and it immediately replaces the state in the current context. A subgoal is evoked by replacing the current goal by the subgoal. Similar replacements set up or change the problem space. The only function not performed is the saving of state, which is automatic, given the assumption of an infinite memory structure.

The horizontal order of the objects in Figure 3-1 is not fortuitous. Each object depends on the ones to its left being established. A problem space is set up in response to a goal; a state functions only as part of a problem space; and an operator is to be applied at a state. When any object changes, all those to its right become functionally irrelevant and need to be determined anew. Therefore, after an object is replaced, all current objects to its right become undefined by a process called *initialization*. Potential inversions of this might be imagined, such as deciding to search for a state from which a given operator could be applied. In fact, such situations must be handled at a higher level of organization. In the case in point, to find the new state in which the desired operator will apply requires applying other operators. Thus, the desired operator cannot be identified simply as the occupant of the current operator slot, but must be identified in some other way, e.g., by association to the goal state (at which it will be applicable).

3.2. Example of the operation of the object structure

We illustrate the basic mechanics of the architecture, ignoring where the control comes from to select each step in Figures 3-2 and 3-3. These figures show a few steps of simple hill climbing. Initially, at time T0, everything is undefined (**undef**), except there is some current goal, G0, and the goal to be attained by hill climbing, G1, is already in the stock. At T1, G1 becomes the current goal. The goal includes an initial state (X0) that is also deposited in the stock. We leave unexplicated the processes that generate from G1 the state X0 and a problem space (P0) at T2; although important, these processes do not play a role in this paper. When the problem space is selected, its operators, Q1 and Q2, are added to the stock of operators. The next step (T3) is for the initial state, X0, to replace **undef** as the current state. Then, at T4, the operator Q1 is

selected from the available admissible operators, to become the current operator. The operator and state are now defined, so that Q1 is applied to X0 to produce X1, a new state. X1 displaces X0 as the current state (T5), and initialization automatically undefines the operator. Thus the search has started up and taken the first step.

Time	Goal	Problem-space	State	Operator
T0	G0 G1	**undef**	**undef**	**undef**
T1	G1 G0	**undef**	**undef** X0	**undef**
T2	G1 G0	P0	**undef** X0	**undef** Q1 Q2
T3	G1 G0	P0	X0	**undef** Q1 Q2
T4	G1 G0	P0	X0	Q1 Q2
T5	G1 G0	P0	X1 X0	**undef** Q1 Q2

Figure 3-2: The operation of the processing structure for Simple Hill Climbing (SHC).

Figure 3-3 continues the behavior, showing the actual hill climbing. The new state (X1) is compared to its *ancestor* state (X0, the state used to generate X1) and the higher state (assume it to be X1) becomes the current state. In this case, no replacement is required because X1 is already the current state. Thus, a step has been taken up the hill, from X0 to X1. At T6, the operator Q1 is selected to be the current operator. As before, since all objects are defined, Q1 is applied to X1 to create a new state (X2) at T7. X2 is compared to its ancestor state, X1 (assume X1 is better than X2). At T8, X1 replaces X2 as the current state. An operator is then selected (T9), and since Q1 has already been applied to X1, Q2 is selected. Q2 is applied to X1 (T10) to create X3 and the operator becomes undefined. X3 is then compared to its ancestor state, X1, and (assume X1 is again better) X1 becomes the current state (T11). A new operator could be selected, but since all available operators have been applied to X1, this state can be extended no further. The current operator is set to **fail** in T12 to signify that no operators are available. Since we are in the context of hill climbing, this failure is viewed as reaching a local maximum (at X1), so the search returns to the supergoal (G0) at T13.

Time	Goal	Problem-space	State	Operator
T6	G1	P0	X1	Q1
	G0		X0	Q2
T7	G1	P0	X2	**undef**
	G0		X0	Q1
			X1	Q2
T8	G1	P0	X1	**undef**
	G0		X0	Q1
			X2	Q2
T9	G1	P0	X1	Q2
	G0		X0	Q1
			X2	
T10	G1	P0	X3	**undef**
	G0		X0	Q1
			X1	Q2
			X2	
T11	G1	P0	X1	**undef**
	G0		X0	Q1
			X2	Q2
			X3	
T12	G1	P0	X1	**fail**
	G0		X0	Q1
			X2	Q2
			X3	
T13	G0	**undef**	**undef**	**undef**

Figure 3-3: Continuation of behavior for Simple Hill Climbing (SHC).

3.3. Search control: The Elaboration-Decision-Application cycle

Search-control knowledge is brought to bear on the process illustrated in Figures 3-2 and 3-3 by means of the *Elaboration-Decision-Application cycle*, which involves three distinct phases of processing. The elaboration phase adds information to the current objects. The decision phase determines which object (goal, problem space, state, operator) is to become current, replacing an existing object in the context. The application phase applies the operator to the state if both are defined. The details of the cycle are shown in Figure 3-4.

The elaboration phase takes objects from the stock as input and augments the current-context objects. Elaboration has access to all the objects in the stock, of whatever type. This includes objects, such as the desired states, operators of the space, etc., that are in the stock by virtue of being augmentations of current objects. It can examine these objects to produce the elaboration. However, the only objects it can affect are those in the current context, which it can augment but not replace.

Elaboration

Inputs:	Objects in stock
Result:	Augments current objects
	Monotone increase in information about current objects
Control:	Continue until quiescent

Decision

Inputs:	Objects in stock
Result:	Votes for, votes against, or vetoes objects in stock
	One new object is selected: It replaces current object of same type
Control:	All voting occurs at the same time

Application

Inputs:	Current state and current operator
Result:	Applies the current operator to the current state
	If a new state results, adds it to the stock and replaces the current state
	(This moves to a new node in the search tree)
Control:	This is primitive relative to the architecture

Figure 3-4: The Elaboration-Decision-Application cycle.

Elaboration accomplishes two related functions. First, an object's unaugmented representation may not make explicit the information necessary for a decision process that is limited in power, such as the decision phase. Second, elaboration can add knowledge about the history of the object during problem solving that will be used later by the decision process. The processes of elaborating this knowledge cannot always operate in a single step. Information made explicit by one item of knowledge in search control, may enable another item of knowledge to make something else explicit. Thus, successive iterations of elaboration (where many elaborations can happen in one step) are possible, until a state of quiescence is reached.

Structurally, the process of elaboration is strictly monotonic. Existing data cannot be modified, only augmented. This does not guarantee that the process is logically monotone; that depends on the content of the search-control knowledge and the objects being elaborated. Likewise, nothing guarantees that the elaboration phase terminates. Indeed, a basic breadth-first resolution theorem-prover would provide an entirely acceptable elaboration process within the specifications so far. However, for present purposes, we assume that the elaboration is relatively short lived, either because of the structure of the knowledge encoded or because of a predefined limit on the duration of any elaboration phase. Though this latter would force the voting processes to act on the basis of incomplete information, it would not induce processing errors at the level of the architecture. A suitable metaphor for elaboration is the coming into awareness of the information

in a complex display under continued perception.

The decision phase follows the elaboration phase and converts the symbolized knowledge accumulated by the elaboration phase into behavior by producing a replacement of some current object. It is a voting procedure with the candidates being the objects in the stock. Votes are registered for each object and the winner replaces the corresponding object in the current context. The decision processes can consult all the objects in the stock. The only outcome is voting; the votes are not encoded or available to other voters. Hence at voting time the situation is static and the votes are simply taken and tallied. Three types of votes are allowed: vote-for, vote-against and veto. Vote-for and vote-against each contribute one vote, respectively plus or minus, for their candidate; a veto ensures that the candidate will never win. The winner of a vote is the unvetoed candidate with the most net votes. If there is a tie in the voting, an arbitrary selection from the tied objects is made (in the particular implementation described in Section 5, the oldest object wins if there is a tie). If no candidate has a net positive vote, a special symbol, **fail**, wins. There is a winner of the voting for each type of object (which can be the existing current object, i.e., the status quo). However, given initialization, only the leftmost object that is changed will survive, all those to its right become undefined.

The application phase follows the decision phase, if all of the context objects are defined (and not fail). The application of the operator to the state is a primitive operation in the architecture, with the current operator and state as inputs and a new result as the output.[7] The new result becomes the current state and the current operator becomes undefined (through initialization). As part of application, an association (one type of augmentation) is built between the operator, state and result. This can be used to determine a state's ancestor and descendents, as well whether an operator has been applied to a state. If the current operator fails to apply, the current state does not change, but an association is built between the operator and the state. After the application phase, the cycle continues by returning to the elaboration phase.

The elaboration and decision phases constitute the search-control knowledge of the agent. We use a specialized *production system* (Waterman & Hayes-Roth, 1978) to implement these phases. Our production system consists of a collection of *productions* of the form:

$$If\ C_1\ and\ C_2\ and\ ...\ and\ C_n\ then\ A$$

The C_i are *conditions* that examine the current object context and the rest of the stock. The form of the conditions is limited to some class of patterns on the encoding of the objects.[8] A is an *action* that either adds knowledge to a current object via an augmentation (for an elaboration production) or casts a vote for an

---

[7] Accomplishing the operator may, of course, require processing in a subspace or even in some representation outside of the architecture.

[8] The exact class of patterns is not important, as long as it is similar to those typical of current production systems.

object (for a decision production). A production is *satisfied* if the conjunction of its conditions is satisfied. There can be any number of productions satisfied at a time. All satisfied elaboration productions fire concurrently and asynchronously during the elaboration phase. All satisfied decision productions fire together during the decision phase. The structure of the architecture (only augmentations, no deletions, etc.) assures that there are no synchronization problems. Thus, the only conflict-resolution principle is *refraction*, which specifies that each instantiation of a production to the stock executes only once.

There is an underlying search that compares the conditions of the productions to the contents of the stock to determine which productions are satisfied. This is the knowledge search, as described in Section 2. The productions are the search-control knowledge, and each relevant item of search-control knowledge must be found. This is much easier than problem solving in general, and efficient architectures can be built to interpret production systems with a large number of productions (Forgy, 1982).

3.4. Example of the operation of search control

Figure 3-5 shows the search-control productions for the simple hill climbing presented in Figure 3-2. The first production is responsible for elaborating the current state with an evaluation of the state's closeness to the goal, a better evaluation meaning that the state is closer to the goal.[9] The rest of the rules are decision productions, responsible for determining the current context. If the current goal succeeds or fails, the two goal-decision productions vote for the supergoal. In this example, the goal is reached when either a specific state or a local maximum in the evaluation function is encountered. In a pure maximization problem, only the second production would be used. The two state-decision productions are the heart of simple hill climbing. The first production votes for the current state if it is better than its ancestor state. This is the knowledge to take a step forward up the hill. The second production votes for the ancestor state if it is better than the current state. This is the knowledge to return to the ancestor state if the newly generated state is not further up the hill. The operator-decision productions veto operators that have already been applied to the state, and they vote for an operator if it will apply to the state. If all operators have been applied to a state without producing a state with a better evaluation, then all operators are vetoed and fail wins.

The SOAR architecture provides the interpreter, I, and the search control productions in Figure 3-5 provide the behavior specification, S. Together, these create the complete control, $[I, S] = C$. This control will produce the behavior described in Figures 3-2 and 3-3 for any operator set Q. Consider Figure 3-2, starting at T3, where X0 has become the current state during the decision phase. Since the current operator is still undefined, there is no application phase. Hence, SOAR goes to the elaboration phase, starting the processing

[9] Computation of the evaluation might require several elaboration productions or it might even, like operators, require processing in another problem space or some other processing system; in these cases this elaboration production is the evoking control.

Elaboration

State: If the current state does not have a evaluation, evaluate the state and augment the state with the result.

Decision

Goal: If the current state matches the desired state, vote for supergoal.
Goal: If the current operator is fail, vote for supergoal.
State: If the current state has an evaluation better than its ancestor state, vote for the current state.
State: If the current state has an evaluation worse than its ancestor state, vote for its ancestor state.
Operator: If an operator has been applied to a state before, veto it.
Operator: If an operator is associated to the current problem space, vote for it.

Figure 3-5: Search control for Simple Hill Climbing.

for T4. The state-elaboration rule is satisfied and is applied, creating an evaluation of the state. No further elaboration productions are satisfied, so the decision phase is entered. The goal-decision productions and the state-decision productions do not contribute, but the second operator-decision production votes for all of the operators in the problem space. The tie is broken arbitrarily, and Q1 wins and becomes the current operator (producing the situation at T4). All context objects are defined, so the application phase is entered. The operator (Q1) is applied to the state, producing X1, which becomes the current state (at T5). In the elaboration phase of T6, X1 is augmented with an evaluation. In the decision phase, X1 is compared to X0, and X1 is found to be better, so the first state-decision production votes for it (the voting does not change the state). This initiates another cycle at T6. No elaboration productions fire, but again an operator is selected in the decision phase, which is then applied in the following application phase (leading to T7). X2 is evaluated and in T8 the second state-decision production is satisfied and votes for X1. X1 replaces X2 and the operator remains undefined. In the following elaboration phase, no productions are satisfied, so the decision phase is entered again. This time an operator is selected. This continues until T11, when both operators receive vetoes, and SOAR inserts **fail** as the current operator (producing T12). In the following decision phase, the second goal-decision production votes for the supergoal, and the supergoal becomes the current goal.

4. A Universal Weak Method

The architecture of the previous section is suitable for encoding many weak methods. We illustrated this only for hill climbing (Figure 3-5), but analogous sets of elaboration and decision productions can be written for other weak methods. To demonstrate that the architecture is suitable for realizing weak methods generally, the obvious path is to write productions for all weak methods, along with a selection mechanism to evoke them. The collection of production systems can then be examined for how perspicuously they encode the weak methods.

There are difficulties with this approach. One is the lack of a complete definition of the set of weak methods. Lists, such as Figure 1-1, contain methods that happen to have become prominent enough to be noticed and named. Moreover, even if a complete list of weak methods were available, the genesis of the methods would still remain -- how did the system come to acquire each member of this list.

4.1. The possibility of a universal weak method

An alternative organization is not to have the methods available as distinct behavior specifications, but to generate each method's behavior as a function of the task situation. The obvious form of such an organization is to analyze the situation and synthesize the appropriate weak method. There is an instructive flaw in such a proposal. Weak methods are used in situations where the agent has little knowledge about the task environment -- as courts of last resort when expertise is missing, and courts of first resort when initial contact occurs and little has yet been found out. The use of a powerful analyzer does not seem consonant with the use of weak methods in either situation. It smacks of the homunculus -- of invoking an intelligent subsystem to support a putatively simple component of an intelligent system.

This suggests that the weak methods should arise out of the structure of the agent and a proposed task. There should exist something, call it a *universal weak method (UWM)*, that together with the agent's knowledge of the task, responds to a situation by behaving according to the appropriate weak method, without the need for significant analysis or synthesis of a behavior specification. A single method as the basis of all weak methods could have important consequences for the acquisition of the weak methods and for the availability of all of them as appropriate. It could also provide a basis for defining the set of all weak methods as those realizable through the universal weak method. It could certainly lay a claim to simplicity and parsimony. Let us explore the possibility of such a method.

A method consists of a representation of behavior, S, and an interpreter, I, for producing that behavior from S. I is now fixed to be SOAR. S is determined by what is assumed about the goal, the task environment and the processing structure of the agent -- what was termed the operational demands. We now seek a factorization of the specification of method M into two parts, $S_U + S_M$, where S_U is common to all weak

methods and S_M is unique to method M. S_U will become the universal weak method and S_M will be the *method increment* for M. By adding to it the S_M associated with any weak method, M, it will produce the behavior of M.

A number of conditions must hold for a proposed factorization to provide a satisfactory universal weak method:

1. The combination of S_M and S_U must produce well-formed methods. (Neither component need be a method in isolation, although either can be.)

2. S_U should specify a nontrivial process; it is easy enough to produce a factorization if little is factored out.

3. S_M and S_U must be combined during problem solving. It must happen easily and quickly, without complex analysis or synthesis; otherwise it will raise issues either of the homunculus or of efficiency.

4. S_M must be extracted from the task environment during problem solving. It must also happen easily and quickly, without complex analysis; otherwise it will raise issues either of the homunculus or of efficiency.

Consider hill climbing. In Section 2 we listed three operational demands: (1) to represent states, (2) to apply operators, and (3) to compare adjacent states to determine which is higher (closer to goal attainment). A possible factorization might assign representing states and applying operators to S_U and comparing adjacent states to S_{HC}. Representing states and operators must then be part of the common specification used (or available) for every weak method. Some information about the factorization can be determined immediately for this choice -- S_{HC} is certainly not a method and S_U is nontrivial -- but the key conditions (such as how S_{HC} and S_U are combined) require a specific organization for the S_M and S_U to be nominated.

4.2. The proposed universal weak method

The weak methods produce the behavior that is possible with some small amount of knowledge about the task environment. Each weak method should then be characterizable by this knowledge. Thus, the method increment, S_M, could be a control process corresponding just to the specific knowledge of the task environment exploited by that weak method. This would imply that S_U would be composed of what is is left after the knowledge characteristic of each weak method per se has been removed. This would be the *default* behavior: What can be done, given that nothing is known about the task environment. In standard architectures, the default behavior is to do nothing, that is, to behave as a computer without a program.[10]

[10] In systems with resident operating systems, the default behavior tends to be going into a quiescent receptive state to await new external input.

However, the default behavior of our agent is dictated by the problem-space hypothesis: To search in a problem space with no search-control information to guide the search. Such a default implies that an agent is always in contact with a task environment by being in some problem space that represents that environment; it always has some representation of states, a set of operators, and a goal that can be translated into goal states in the space. Thus, a candidate factorization is:

1. *The Universal Weak Method (UWM)* is the default method of search in a problem space with no search control.

2. Individual weak methods are method increments (S_M), formed from the incremental knowledge of the task environment that is demanded by the weak method.

Figure 4-1 gives the elaboration and decision productions that constitute S_U, the UWM. The first elaboration production detects if all problem spaces have been vetoed and marks the goal unacceptable. The goal must be elaborated with this information so that when it is no longer the current goal, it will not be selected as a subgoal. The other two elaboration productions serve the same purpose for problem spaces and states. In the case of states, this signifies that all of the available operators have been exhausted for the marked state. All the decision productions for the goal, problem space, and state vote for an acceptable object (i.e., one not marked unacceptable) and veto unacceptable objects. These allow all acceptable candidate objects to be considered, independent of any method or task specific knowledge. The operator-decision productions perform the same task for operators.

Elaboration:

Goal: If the current problem space is **fail**, the goal is unacceptable.
Problem Space: If the current state is **fail**, the problem space is unacceptable.
State: If the current operator is **fail**, the state is unacceptable.

Decision:

Goal: If there is an acceptable available goal, vote for it.
Goal: If there is an unacceptable available goal, veto it.
Problem Space: If an acceptable problem space is associated to the current goal, vote for it.
Problem Space: If an unacceptable problem space is associated to the current goal, veto it.
State: If an acceptable state is in the current problem space, vote for it.
State: If an unacceptable state is in the current problem space, veto it.
Operator: If an acceptable operator is associated to the current problem space, vote for it.
Operator: If an operator has already been applied to the current state, veto it.

Figure 4-1: The universal weak method (UWM).

Let us examine the factorization conditions. First, S_U is actually a particular method in its own right, namely search without any heuristics for state or operator selection. S_M, being the modification of heuristic search to incorporate particular knowledge, will generally not be a method. Second, the UWM specifies a nontrivial behavior, even though it is a very simple specification that provides just enough control to search a

problem space. The simplicity of Figure 4-1 derives in large part, of course, from the problem-solving character of the underlying SOAR architecture.

Third, the weak methods are to be formed by adding additional specifications, S_M, to S_U. The composition is performed by adding the elaboration and decision productions that correspond to the increments of knowledge that constitute the operational and effectiveness demands of a particular weak method. For instance, if the state-elaboration and state-decision productions from Figure 3-5 are added to the productions for the UWM (Figure 4-1), then the combined set of productions performs simple hill climbing (SHC). The hill-climbing state-decision productions will dominate the state-decision productions of the UWM because they always vote for a subset of the states that the UWM votes for. Thus, adding the increment of control information to the universal weak method to produce the actual weak method is easily accomplished. It requires only the addition of productions to production memory.

Fourth, the incremental productions are to be extracted from the task environment easily and quickly. This condition cannot be fully addressed without an explicit model of task-environment knowledge to define the extraction process. We will not present such a model in this paper. However, the major ingredient for satisfying this condition is already present, namely, that the method increments are factored cleanly from the UWM, so they consist of productions that deal only with the incremental knowledge of the task environment that defines the weak method. This can be seen in hill climbing, where the incremental productions are concerned exclusively with the knowledge that is specific to hill climbing, namely, the evaluation function and what it implies for behavior. Nothing in the method-increment productions involves other control aspects of the method.

4.3. Method Increments

The productions that form the method increments (S_M) can be split into two functional types: elaboration and decision productions. The elaboration productions are local to the current task and create knowledge about specific states and operators. These create a set of *concepts* about the task, such as an evaluation of a state, whether the state is a duplicate state, or the depth of search to the state. These concepts embody the operational demands of the method. Thus in hill climbing, it must be possible to obtain the evaluation of a state or the method cannot be applied. However, the concepts do not specify how behavior is to be affected; they only provide the basis for specification. The productions that compute these concepts are generally task dependent and must be provided separately for each problem space. Exceptions can occur when the concept concerns the behavior of the problem solver, which is the same no matter what specific task is being performed, e.g., the depth of search to a state.

The decision productions convert the knowledge embodied in these concepts into action by testing concepts

and making votes. The decision productions contain task-independent knowledge. More precisely, they depend on the task environment only through the concepts provided by the elaboration productions and these same concepts may be computed on many different tasks. However, the decision productions have conditions that tie them to specific tasks (goals or problem spaces). When the agent has a decision production for a task, that production is the agent's knowledge that the behavior the production produces (through votes) is appropriate to the given task.

The productions of the UWM actually provide an interesting example of the two functions. The UWM elaboration productions define a basic notion of the concept of *unacceptable* in terms of the symbol **fail**, which is itself defined by the architecture. The UWM decision productions are all concerned with converting the concept of unacceptable (and acceptable) into action. Thus the UWM productions as a whole provide the semantics of the concept of acceptability. Method increments can make use of this concept for their own purposes, but they must respect the meaning of acceptability as defined by the UWM, because the method-increment productions will be added to those of the UWM and cannot negate their actions.

5. Experimental Demonstration

In this section we demonstrate empirically that the universal weak method just defined is capable of producing many weak methods. We cannot speak of all the weak methods, because (as noted earlier) not only is the set not yet well-defined, but we hope ultimately to define it in terms of the universal weak method itself. We will restrict the scope of the demonstration even further by not considering methods that involve subgoals. This restriction will be discussed in the next section. It rests on the (predicted) existence of another functional capacity of a general intelligent agent, which we call *universal subgoaling*, to set up subgoals to cope with difficulties that arise in accomplishing a task. Universal subgoaling raises many issues about which we have only preliminary understanding currently, and which require a separate treatment in any event. We expect that the universal weak method and universal subgoaling jointly will produce all that might reasonably be called weak methods. Thus, in this paper we can only address methods that do not involve subgoals. This still permits a significant demonstration.

5.1. Conditions for demonstrating a universal weak method

To demonstrate the proposed universal weak method requires the following:

1. Implement a system that incorporates the SOAR problem solving architecture of Section 3.

2. Encode the universal weak method in that system, corresponding to the productions of Figure 4-1.

3. Encode a set of tasks for the system. A task is defined by a problem space and goal states within the problem space. We need not deal with the processes for creating problem spaces and goal states, since a universal weak method operates with these as given.

4. Run each task with the universal weak method by itself, that is, with the default heuristic search with no task-specific search-control knowledge. This shows both that the task is encoded successfully and that the universal weak method works as a weak method in its own right.

5. Encode a set of weak methods, corresponding to the productions for simple hill climbing given above. These will necessarily remain incomplete to the extent they contain knowledge of a specific task environment that cannot be determined until the specific task is given.

6. Run each weak method on each task, by adding the weak-method incremental productions and the requisite task-environment specific productions to the fixed set of productions comprising the universal weak method and the fixed encoding of the problem space and goal-state recognition. The addition is simply to take the set union of all the productions. The weak methods will not be applicable to all tasks, but only to those tasks that admit of the knowledge required by the method.

7. Verify that the behavior of the system constitutes that of the weak method. The effectiveness of the behavior, which depends on the knowledge in the method, is irrelevant; what counts is only whether behavior appropriate to the method is produced.

5.2. The production system and the tasks

The problem solving architecture is implemented in a production system architecture, XAPS2 (Rosenbloom & Newell, 1982), which was chosen because all its satisfied productions fire in parallel, supporting the parallel features and the voting of the elaboration and decision phases.[11] The details of this implementation are not critical and we shall continue to use the descriptions at the level of Figure 4-1. Full details on all tasks, methods and runs can be found in Laird (1983).

Figure 5-1 gives brief descriptions of the fourteen tasks used for the demonstration. Included in the description is a statement of the task, followed by a description of the states and operators that define the problem spaces we chose for the tasks. For the Picnic Problem and the Root Finding task, more than one problem space was implemented. The tasks are simple illustrative tasks, familiar from the AI literature, with a few even simpler decision and logical tasks. Such tasks are suitable for an initial test of a universal weak method, being familiar, knowledge-lean and relatively easy to implement. Most of the tasks require searches of combinatorial spaces, where state or operator selection determines the success of a method. A few have monotonic problem spaces where the operator selection is critical.

Each of these tasks is encoded as a set of productions. There are productions for the operators of each problem space and for detecting the goal states. These productions define representations of the states in the problem space. For instance, the Eight Puzzle encodes the state as a 3x3 grid of cells *[cell (i,j), tile]*, with eight tiles, 1, ... 8, and the 0 tile, indicating the empty cell. There is an operator for each possible movement of tiles into the empty cell (there are 24 such moves). Thus, operator (2,2)→(3,2) moves the tile in cell (2,2) into the empty cell, which is (3,2). Each of these operators is realized by a production (operators for some tasks take two productions). Operators that have multiple instantiations for a given state are possible. Each instantiation must be added to the stock (as an elaboration of the current state) with a unique name, and the operator productions will implement a class of operators instead of a single one. The initial state is a particular configuration, the final state is encoded by another state, and there is a production that detects if a state matches the final state. These productions are simply added to the productions for the universal weak method to form the basic knowledge of the task.

Figure 5-2 shows the initial few moves for the default behavior of the universal weak method in isolation on the Eight Puzzle. Above each state is the operator used to create it. Below the operator is the name of the state the operator was applied to, followed by the resulting state's name (S1⇒S2). The initial state of the problem is S1. The desired state has the numbers 1-8, in order, around its border, starting from the upper left

---

[11] We have since implemented the system in OPS5 (Forgy, 1981) with a modified conflict resolution to provide the required concurrency.

Eight Puzzle
Problem Statement: There are eight numbered movable tiles set in a 3x3 frame. One cell is always empty, making it possible to move an adjacent numbered tile into the empty cell. The problem is to transform one configuration to a second by moving tiles.
States: States are configurations of the numbers 0-8 in a 3x3 grid.
Operators: There are twenty-four operators, one for each possible movement of tiles in cells adjacent to the empty cell into that cell.

Tower of Hanoi
Problem Statement: A board has three pegs, 1, 2 and 3. On Peg 1 are three disks of different sizes, in order with the largest at the bottom. The task is to get all the disks on Peg 3 in the same order. A disk may be moved from any peg to another, providing that it is the top disk on its peg and that it is not put on top of a disk smaller than itself.
States: Arbitrary configurations of the three disks on the three pegs.
Operators: There are six operators that move the top disk on a peg to another peg.

Missionaries and Cannibals
Problem Statement: Three missionaries and three cannibals wish to cross a river. Though they can all row, they only have available a small boat that holds two people. The difficulty is that the cannibals are unreliable; if they ever outnumber the missionaries on a river bank, they will kill them. How do they manage the boat trips so that all six get safely to the other side?
States: The states contain the number of missionaries and cannibals on each side of the river and the position of the boat.
Operators: There are ten operators, one for each legal combination for moving missionaries, cannibals, and boat across the river.

Water Jug Problem
Problem Statement: Given a five-gallon jug and a three-gallon jug, how can precisely one gallon of water be put into the three-gallon jug? Since there is a well nearby, a jug can be completely filled or completely emptied at will. No measuring devices are available other than the jugs themselves.
States: The states contain the amount of water in the five gallon jug and the amount of water in the three gallon jug.
Operators: There are six operators, one for each combination of pouring water between the well and the two jugs.

Picnic Puzzle
Problem Statement: Al, Bill and Chris planned a big picnic. Each boy spent 9 dollars. Each bought sandwiches, ice cream and soda pop. For each of these items the boys spent jointly 9 dollars, although each boy split his money differently and no boy paid the same amount of money for two different items. The greatest single expense was what Al paid for ice cream. Bill spent twice as much for sandwiches as for ice cream. All amounts are in round dollars. How much did Chris pay for soda pop?
States: The three formulations (see below) have similar problem spaces with states that contain the amount each boy spent for each item. The constraints are equations and inequalities over the items. All the implementations assume that the maximum cost of any item is 6 dollars (derived from the problem statement).
Note: The problem is from (Polya, 1962) and the problem spaces are adapted from (Newell, McDermott & Forgy, 1977).

Picnic Puzzle I
States: The states contain a 3x3 array with values from 1 to 6, but are viewed as a 9 digit number from 111,111,111 to 666,666,666.
Operators: There is one operator: Add 1 to the 9 digit number and carry if a value is greater than 6.

Picnic Puzzle II
States: The states are a 3x3 matrix of numbers from 1 to 6, or unknown.
Operators: There are six operators. Each operator assigns a number (1-6) to one of the unknown positions in the matrix. The operator instantiations are determined by a predetermined ordering of the positions in the matrix.

Picnic Puzzle III
States: The states are a 3x3 matrix of numbers from 1 to 6.
Operators: There are nine operators, one for each position in the matrix. Each operator adds one to the current value of the position.

Labeling Line Drawings
Problem Statement: There is a line drawing of 3D objects made of trihedral vertices. The problem is to determine a single, consistent labeling of the lines and junctions as edges and vertices of the 3D objects.
States: The states consist of the drawing with sets of possible labelings of the vertices and edges.
Operators: There is one operator: Mark as inconsistent a vertex labeling if one of its line labelings is not available in one of the adjacent consistent vertex labelings.

Figure 5-1: Tasks used for demonstration of the universal weak method (initial part).

Syllogisms
Problem Statement: A syllogistic reasoning task is formed from three terms (e.g., A, B, C), combined into two assertions (the major and minor premise) involving pairs of terms (A and B; B and C), from which some assertion (the conclusion) about the third pair (A and C) may or may not follow. Four logical assertions are possible from the two quantifiers (all, some), negation (no/not) and the copula of implication (are). For example, from *All A are B* and *No B are C*, does it follow that *Some A are not C?*
States: Sets of abstract objects. Each abstract object represents the possible or necessary existence of objects with or without terms A, B and C.
Operators: There are four encoding operators that transform the English statements into sets of abstract objects. There are three combining operators that create new abstract objects for existing abstract objects.
Note: The problem space is adapted from (Newell, 1980b).

Wason Verification Task
Problem Statement: There are four cards on a table. Each card has a number on one side and a letter on the other side. The task is to select those cards that must be turned over to prove that a given logical rule is true for all cards. For example, the cards might be [E], [K], [4] and [7], with the rule: All cards that have a even number on one side have a vowel on the other side.
States: The states consist of the four cards, augmented with a list (possibly empty) of cards to examine in detail.
Operators: There is an operator to turn each card.
Note: This problem has been much investigated (Wason & Johnson-Laird, 1972), because it appears to be difficult for humans.

Simple String Matching
Problem Statement: There are two strings consisting of constants and variables. The problem is to determine if the two strings can be transformed into idential expressions using variable substitution.
States: The states are pairs of strings with pointers to the current elements of the strings being considered.
Operators: There are two operators: Consider the next element in the strings: and Substitute a constant for a variable in all occurrences.

Three Wizards Task
Problem Statement: Long ago a wicked king was searching for a new wizard with whom to plot some devious schemes. He summoned to him three wizards who seemed especially promising, and let them into a small room, which was barren except for a lighted candle on a table in the middle of the room. "Listen to me well," he said. "In a few minutes all of you will be blindfolded and I will paste upon each of your foreheads a uniformly colored spot of black or white paper. At least one spot will be white. The first of you who guesses the color of his own spot will become my new wizard, and ride in his own chariot with all expenses paid. The other two of you will be sent to a terrible fate that I shall not describe. None of you will be allowed to remove any of the spots, and you will each be allowed only one guess." The king then ordered his guards to blindfold the wizards, proceeded to paste white spots on all the wizards' foreheads, and finally had their blindfolds removed. After a few seconds, one of the wizards correctly identified the color of the spot on his forehead. How did he know it?
States: The states represent what a wizard knows. This includes whether a wizard has guessed, the color of another wizard's patch, how the color of the wizard's patch was deduced, whether there is a conflict in the knowledge, and what another wizard knows (which can include all of the above). This last part of the state is recursive and leads to some of the difficulty with the problem.
Operators: There are eight operators that add knowledge to the state: (1) Consider what another wizard knows; (2) Ascribe public knowledge to what another wizard knows; (3) Assign a specific value to an unconfirmed patch; (4) Deduce that if X knows that the other two wizards have black patches, he knows he must have a white one; (5) Deduce that if X knows that Y knows something, then X knows it; (6) Deduce that if X knows the color of his patch, he should guess: (7) Deduce that if X should guess and has not guessed, there is a conflict: (8) Deduce that if there is a conflict in X, the most recent assignment is incorrect and the other value is confirmed.
Note: The problem space is adapted from (Newell, McDermott & Forgy, 1977).

Root Finding I
Problem Statement: Given a polynomial equation in terms of X, find a root of the equation. For example: $Y = 4X^2 - 10X + 4 = 0$.
States: The states contain the current guess (X) and prior guess (X_{prior}) as well as the absolute value of the equation evaluated with those guesses (Y and Y_{prior}).
Operators: There are three operators: (1) Make the current guess be $2X - X_{prior}$; (2) Make the current guess be $X + (X - X_{prior})/2$; (3) Make the current guess be $X - (X - X_{prior})/2$.

Root Finding II
Problem Statement: Given a polynomial equation in terms of x, find a root of the equation. For example: $Y = 4X^2 - 10X + 4 = 0$.
States: The states contain the current guess (X) and prior guess (X_{prior}) as well as the absolute value of the equation evaluated with those guesses (Y and Y_{prior}).
Operators: There is only one operator, compute a new guess using Newton's formula: $X_{new} = (YX_{prior} - XY_{prior})/(Y - Y_{prior})$.

Figure 5-1: Tasks used for demonstration of the universal weak method (continued).

corner. The selection of operators is made arbitrarily among the legal ones. This happens to result in a breadth-first search due to features of the underlying architecture. If the search were to stumble across the desired state, it would recognize it. All of the fourteen tasks in Figure 5-1 show similar default behavior.

	(2,2)→(3,2)	(3,1)→(3,2)	(3,3)→(3,2)	(1,2)→(2,2)	(2,1)→(2,2)	(2,3)→(2,2)
S1	S1⇒S2	S1⇒S3	S1⇒S4	S2⇒S5	S2⇒S6	S2⇒S7
2 8 3	2 8 3	2 8 3	2 8 3	2 0 3	2 8 3	2 8 3
1 6 4	1 0 4	1 6 4	1 6 4	1 8 4	0 1 4	1 4 0
7 0 5	7 6 5	0 7 5	7 6 0	7 6 5	7 6 5	7 6 5

Figure 5-2: Behavior of the universal weak method only (default behavior) on the Eight Puzzle.

5.3. The S_M increments for the weak methods

Figure 5-3 gives the twelve weak methods that occur in the demonstration. One method, avoid duplication (AD) does not occur in Figure 1-1, because it is usually simply incorporated as an additional mechanism in other methods. Some methods (DFS and BFS) each have two alternative versions that produce identical behavior (both versions were run). Two weak methods, generate and test (GT) and match (MCH), showed up in the demonstration, but without special productions (S_M). These methods will be discussed later. The weak methods in the figure (excepting AD) correspond to the subset of those in Figure 1-1 that need not involve subgoals.[12] With each method is given the task-independent productions of the method that are added to the universal weak method (S_U) to produce the behavior of the weak method. These are all either decision productions, or elaboration productions that produce concepts that are independent of a task (such as depth). The task-dependent concepts used in each weak method are in ***bold-italics***. To complete the method, task-dependent elaboration productions had to be added to compute each such concept. In addition, in actual runs, a task-dependent decision production was added to vote for operators that would apply to the current state. All methods would have worked without this, but it is a useful heuristic that saves uninteresting search where an operator is selected, does not apply, and is then vetoed because it was attempted for the current state.

Examination of each of the method increments in the figure confirms the claim made in Section 4.1 in regard to the fourth factorization condition, namely, that method increments do not specify any control that is not directly related to the task-environment knowledge that specifies the weak method. All the control (the decision productions) deals only with the consequences of the concepts for control of the method behavior. The missing task-dependent productions do not involve any method-control at all, since they are exclusively devoted to computing the concept.

---

[12] On the other hand, any weak method can be implemented using subgoals to perform some of its decisions, such as operator selection.

Avoid Duplicates (AD)
State: If the current state is a ***duplicate*** of a previous state, the state is unacceptable.
Operator: If an operator is the ***inverse*** of the operator that produced the current state, veto the operator.
Operator-Selection Heuristic Search (OSHS)
State: If the state ***fails*** a goal constraint, it is unacceptable.
Operator: (These are task-specific decision productions that vote for or vote against an operator based on the current state.)
Means-Ends Analysis (MEA)
Operator: If an operator ***can reduce*** the ***difference*** between the current state and the desired state, vote for that operator.
Breadth-First Search (BrFS)
State: If a state's depth is not known, its depth is the depth of the ancestor state plus one.
State: If a state has a depth that is not larger than any other acceptable state, vote for that state.
Depth-First Search #1 (DFS)
State: If the current state is acceptable, vote for it.
State: If the current state is not acceptable, vote for the ancestor state.
Depth-First Search #2 (DFS)
State: If a state's depth is not known, its depth is the depth of the ancestor state plus one.
State: If a state has a depth that is not smaller than any other acceptable state, vote for that state.
Simple Hill Climbing (SHC)
State: If the current state is not acceptable or has a ***evaluation*** worse than the ancestor state, vote for the ancestor state.
State: If the current state is acceptable and has a ***evaluation*** better than the ancestor state, vote for the current state.
Steepest Ascent Hill Climbing (SAHC)
State: If the ancestor state is acceptable, vote for the ancestor state.
State: If the current state is not acceptable and a descendent has an ***evaluation*** that is not worse than any other descendent, vote for that descendent.
State: If the current state is acceptable, and the ancestor state is not acceptable, vote for the current state.
Best-First Search #1 (BFS)
State: If a state has an ***evaluation*** that is not worse than any other acceptable state, vote for that state.
Best-First Search #2 (BFS)
State: If a state has an ***evaluation*** that is better than another state, vote for that state.
Modified Best-First Search (MBFS)
State: If the ancestor state is acceptable, vote for the ancestor state.
State: If the current state is not acceptable, and a state has a ***evaluation*** that is not worse than any other state, vote for that state.
State: If the current state is acceptable, and the ancestor state is not acceptable, vote for the current state.
A*
State: If a state's ***estimated distance to the goal*** plus its depth is not larger than any other state, vote for that state.
MCH
Note: No incremental productions; method arises from the structure of the task.
GT
Note: No incremental productions; method arises from the structure of the task.

Figure 5-3: Method increments: The search control for weak methods

Figure 5-4 shows the moves for two weak methods on the Eight Puzzle. The first is simple hill climbing (SHC); the second is depth-first search (DFS). The information added was just the productions in Figure 5-3 for each method. The hill climbing productions had to be instantiated for the particular task because the evaluation function is task-specific. A simple evaluation was used, namely the number of tiles already in their correct place. Additional elaboration productions had to be added to compute this quantity for the current state. Depth-first search, on the other hand, requires no task-specific instantiation, because it depends on an aspect of the behavior of the agent (depth) that is independent of task structure. Examination of the traces will show that the system was following the appropriate behavior in each case.

Adding the simple hill-climbing knowledge to the UWM allows SOAR to solve the problem.[13] A typical problem solver using simple hill climbing would reach state S2, find it to be a local maximum and terminate the search. With the UWM, S2 is also found to be a local maximum, and simple hill climbing does not help to select another state. However, the default search control still contributes to the decision process so that all states receive votes (although S2 is vetoed). To break the tie, SOAR choses one of the states as a winner. The oldest state always wins in a tie (because of a "feature" in the architecture) and S1 is selected. S7 and S8 are generated, but they are no better than S1. S1 is now exhausted and receives a veto. Another local maximum is reached, so the default search control comes into play again. The oldest, unexhausted state is S3 and it generates S9 which is better than S3. The hill climbing productions select states until the desired state is reached (S11). Depth-first search alone (without AD) does not solve the problem because it gets into a cycle after 12 moves and stays in it because duplicate states are not detected.

```
Simple Hill Climbing (SHC)
            (2,2)→(3,2) (1,2)→(2,2) (2,1)→(2,2) (2,3)→(2,2) (3,2)→(2,2) (3,1)→(3,2)
   S1          S1⇒S2       S2⇒S3       S2⇒S4       S2⇒S5       S2⇒S6       S1⇒S7
2 8 3          2 8 3       2 0 3       2 8 3       2 8 3       2 8 3       2 8 3
1 6 4          1 0 4       1 8 4       0 1 4       1 4 0       1 6 4       1 6 4
7 0 5          7 6 5       7 6 5       7 6 5       7 6 5       7 0 5       0 7 5

            (3,3)→(3,2) (1,1)→(1,2) (2,1)→(1,1) (2,2)→(2,1)
               S1⇒S8       S3⇒S9       S9⇒S10      S10⇒S11
               2 8 3       0 2 3       1 2 3       1 2 3       success
               1 6 4       1 8 4       0 8 4       8 0 4
               7 5 0       7 6 5       7 6 5       7 6 5

Depth-First Search (DFS)
            (3,1)→(3,2) (2,1)→(3,1) (2,2)→(2,1) (3,2)→(2,2) (3,1)→(3,2)
   S1          S1⇒S2       S2⇒S3       S3⇒S4       S4⇒S4       S5⇒S6
2 8 3          2 8 3       2 8 3       2 8 3       2 8 3       2 8 3       cycles
1 6 4          1 6 4       0 6 4       6 0 4       6 7 4       6 7 4
7 0 5          0 7 5       1 7 5       1 7 5       1 0 5       0 1 5
```

Figure 5-4: Behavior of Simple Hill Climbing and Depth-First Search on the Eight Puzzle.

5.4. Results of the demonstration

Figure 5-5 gives a table that shows the results of all the weak methods of Figure 5-3 against all the tasks of Figure 5-1, using the incremental productions specified in Figure 5-3. In all the cases labeled +, the behavior was that of the stipulated weak method. In the cases left blank there did not seem to be any way to define the weak method for the task in question. In the case of methods requiring state-evaluation functions, an evaluation function was created only if it had heuristic value. Although in principle there could be an issue of determining whether a method is being followed, in fact, there is no doubt at all for the runs in question. The

[13] Simple hill climbing may often lead the search astray in other Eight Puzzle problems. But, again, the effectiveness of these methods is not at issue, only whether the universal weak method can produce the appropriate method behavior.

structure of the combined set of productions makes it evident that the method will occur and the trace of the actual run simply serves to verify this.

Task	UWM	AD	OSHS	MEA	BrFS	DFS	SHC	SAHC	BFS	MBFS	A*
Eight Puzzle		+	+	+	+	+	+	+	+	+	+
Tower of Hanoi	+	+	+		+	+					
Missionaries	+	+	+	+	+	+	+	+	+	+	+
Water Jug	+	+	+		+	+					
Picnic I	+										
Picnic II	+	+	+	+	+	+	+	+	+	+	+
Picnic III	+	+	+	+	+	+	+	+	+	+	+
Labeling[14]	+		+								
Syllogisms	+			+							
Wason	+		+								
String match[14]	+			+							
Wizards	+		+								
Root Finding I	+	+	+		+	+	+		+		
Root Finding II	+										

Figure 5-5: All methods versus all tasks

All fourteen tasks in Figure 5-5 are marked under UWM, indicating that they were attempted (and sometimes solved) without additional search control knowledge. For Picnic I, there is no additional search control knowledge available, so this is the only method that can be used with it. With Root Finding II, the problem (given the problem space used) is simple enough so that no search control knowledge is necessary to solve it. Four of the tasks (Eight Puzzle, Missionaries, Picnic II and III) can use a variety of search control knowledge to both select operators and states so that it was possible to use all methods for these tasks. Tower of Hanoi, the Water Jug puzzle, and the Wason task depend on operator selection methods (and avoiding duplicate states) to constrain their searches. The Syllogism and Wizards task are monotonic problems, where all of the operators add knowledge to the states, so that state selection is not an issue. However operator selection is needed to avoid an exponential blowup in the number of operator instantiations that would occur if some operators were applied repeatedly.

[14] These tasks were run on a successor architecture to SOAR, being developed to explore universal subgoaling; however, it has the same essential structure with respect to the aspects relevant here.

As mentioned earlier, two weak methods occurred during the demonstration, generate and test, and match, that did not require any incremental productions. In Picnic I, each state is a candidate solution that must be tested when it is created. The UWM carries out the selection and application of the single operator to create the new states, producing a generate-and-test search of the problem space. In the Simple String Match, which is a paradigm case for MCH, the UWM itself suffices to carry out the match, given the operators. The special knowledge that accompanies MCH that failure at any step implies failure on the task (so that backing up to try the operators in different orders is futile), is embedded in the definition of the problem space. The two examples simply illustrate that the structure of the problem space can play a part in determining the method.

6. Discussion

A number of aspects of the SOAR architecture and the universal weak method require further discussion and analysis.

6.1. Subgoaling

In demonstrating the universal weak method, we excluded all methods that used subgoals, on the grounds that a more fundamental treatment using universal subgoaling was required. Although such a treatment is beyond the scope of this paper, a few additional remarks are in order.

The field has long accepted a distinction, originally due to Amarel (1967), between two fundamentally different approaches to problem solving, the *state-space approach* and the *problem-reduction approach* (Nilsson, 1971). The first is search in a problem space. The second is the use of subgoal decomposition, as exemplified in *AND/OR* search. This separation has always been of concern, since it is clear that no such sharp separation exists in human problem solving or should exist in general intelligence. The SOAR architecture is explicitly structured to integrate both approaches. Goal changes occur within the same control framework as state and operator changes, and their relation to each other is clear from the structure of the architecture. In this paper we have been concerned only with one specific mode of operation: goals set up problem spaces within which problem search occurs to attain the goals. We need to consider modes of operation that involve the goal structure.

New subgoals can be created at any point in searching a problem space. This requires determining that a subgoal is wanted, creating the subgoal object and voting the subgoal to become the current goal object. The new subgoal leads (in the normal case) to creating a problem space, and then achieving the goals by search in the subspace. Reinstating the original goal leads (again, in the normal case) to reinstating the rest of the object context and then extracting the solution objects of the subgoal from the stock. We have not described the mechanics of this process, which is the functional equivalent of a procedure call and return. It is not without interest, but can be assumed to work as described. Tasks have been run in SOAR that use subgoaling.

The normal mode of operation in complex problem solving involves an entire goal hierarchy. In SOAR, this takes the form of many goals in the stock, with decisions about suspending and retrying goals being made by search control voting to oust or reinstate existing goals. The evaluations of which goal to retry are made by the same elaboration-decision cycle that is used for all other decisions, and they are subject to the same computational limitations. Similarly, methods that involve subgoals are encoded in search control directly. Such a method operates by having search control vote in subgoals immediately, rather than apply an operator to take a step in the problem space. How such explicit methods are acquired and become part of search

control knowledge is one more aspect that is beyond this paper. All we have pointed out here is how goal decomposition and problem-space search combine into a single integrated problem-solving organization.

In general, subgoals arise because the agent cannot accomplish what it desires with the knowledge and means immediately at hand. Thus, a major source for subgoals are the *difficulties* that can be detected by the agent. Operator subgoaling, where the difficulty is the inability to apply a selected operator, is the most well known example; but there are others, such as difficulties in selecting operators or difficulties in instantiating partially specified operators. A complete set of such difficulties would lay the base for a universal subgoaling scheme. Subgoals would be created whenever the pursuit of a goal runs into difficulties. Such subgoals would arise at any juncture, rather than only within the confines of prespecified methods. Universal subgoaling would complement the universal weak method in that both would be a response to situations that are novel and where, at least at the moment of encounter, there is an absence of extensive search-control knowledge.

Central to making subgoaling work is the creation of problem spaces and goal states. Every new subgoal requires a problem space. No doubt many of these can pre-exist in a sufficiently well developed form so that all that is required is an instantiation that is within the capabilities of search control. But more substantial construction of problem spaces is clearly needed as well. The solution that flows from the architecture, and from the mechanism of universal subgoaling just sketched, is to create a subgoal to create the new problem space. Following out this line to problem spaces for creating problem spaces is necessary for the present architecture to be viable. There are indications from other research that this can be successfully accomplished (Hayes & Simon, 1976), but working that out is yet one more task for the future.

6.2. The voting process

The architecture uses a voting scheme, which suggests that search control balances off contenders for the decision, the one with the preponderance of weight winning.[15] However, a voting scheme provides important forms of modularity, as well as a means to adjudicate evidence. In a voting scheme, sources of knowledge operate independently of each other, so that sources can be added or deleted without knowing what other sources exist or their nature, and without disrupting the decision process (although possibly altering its outcome). If knowledge sources are highly specialized, so that only a very few have knowledge relevant to a given decision, then a voting scheme is more an opportunity for each source to control the situation it knows about than an opportunity to weigh the evidence from many sources. The balancing aspect of voting then becomes merely a way to deal with the rare cases of conflicting judgement, hopefully without strongly affecting the quality of the decisions.

---

[15] Indeed, a common question is why we don't admit varying weights on the votes of decision productions.

The weak methods and the tasks used for the demonstration provide a sample of voting situations that can be used to explore the functions that voting actually serves in SOAR. We can examine whether the voting was used to balance and weigh many different sources of knowledge (so that the number and weight of votes is an issue), or whether the productions are highly specialized and act as *experts* on a limited number of decisions. The situations are limited to state and operator changes, that is, search within a problem space, with no goal or problem-space changes, but the evidence they provide is still important. The following analysis is based on an examination of the productions used for each method and the traces of the runs of each task using the different methods.

The structure of the state changes is simple. The UWM votes for all acceptable states and vetoes all unacceptable states. All of the weak methods have at most one state-change voting production active at a time. It may vote for many states, but the states will all be equivalent to the method. Thus, voting on states fits the expert model, in which balancing of votes does not occur.

For operator changes, the result is the same (the voting fits the expert model), but the analysis is a bit more complicated. The UWM votes for all operators in the current problem space and vetoes all operators that have already applied to the current state. The weak methods may have many operator-change voting productions active at a time. However, the final winner receives a vote from every production that was voting for an operator[16] and no votes from a production voting against, or vetoing an operator. There may be many operators receiving the same number of votes, but they have votes from the same productions, and the final selection is made randomly from this set. There is never a balancing of conflicting evidence for the final decision. Each time a production is added, it only refines the decision, by shrinking the set from which the operator is randomly selected. Therefore, the weighting and balancing of votes would not affect the selection of operators.

6.3. Unlimited memory

We have assumed an unlimited memory capacity, primarily to simplify the investigation. In fact, each method is characterized by a specific demand for memory. If the memory available is not sufficient for the method, or if it makes disruptive demands for access, then the method cannot be used. A useful strategy to study the nature of methods is to assume unlimited memory capacity and then investigate which methods can be used in agents with given memory structures. Figure 6-1 gives the memory requirements for the stock of the weak methods used in the demonstration. This is the requirement beyond that for the current context, including the space for the problem space and operators. Several of the methods require an unbounded stock

---

[16] Actually, the final winner receives a vote only from every concept, rather than every production, since it is sometimes necessary to implement a concept with more than one production, because of the limited power of the production-system language.

for states. They differ significantly in the rate at which the stock increases and in the type of memory management scheme needed to control and possibly truncate the stock if memory limits are exceeded.

Method	Stock capacity requirement
Universal Weak Method (UWM)	None
Avoid Duplication (AD)	Unbounded: All states visited
Means-Ends Analysis (MEA)	None
Breadth-First Search (BrFS)	Unbounded: The set of states at the frontier depth [equal to the (branching-factor)^{depth}]
Depth-First Search (DFS)	Unbounded: The set of states on the line to the frontier (equal to the depth)
Simple Hill Climbing (SHC)	One: The ancestor state
Steepest Ascent Hill Climbing (SAHC)	Finite: The next-states corresponding to each operator, plus the ancestor state; Or two, for the next-state and the best-so-far
Best-First Search (BFS)	Unbounded: The set of all acceptable states
Modified Best-First Search (MBFS)	Unbounded: The set of all acceptable states
A*	Unbounded: The set of all acceptable states

Figure 6-1: Memory requirements of the methods on the stock, beyond the current context.

Capacity limits on the stock affect how well the agent solves problems, but they do not produce an agent that cannot function, i.e., cannot carry out some search in the problem space. Restricting the selection of states to a small number would cause methods such as best-first search to become more like hill climbing. Problem solving would continue, but the effectiveness of the method might decrease. However, the architecture is not similarly protected against all capacity problems. In particular, if the memory is too limited for the problem space to be represented (e.g., because of the number of operators in the problem space), then the agent cannot function.

6.4. Computational limits and the uniqueness of the UWM

Processing at a state in the problem space must be computationally limited, both for speed (it lies in the inner loop of the search through the problem space) and for functionality (it must be the unintelligent subprocess whose repeated occurrences give rise to intelligent action). Neither of these constraints puts a precise form on the limitation, and we do not currently have a principled computational model for this processing. Indeed, there may not be one. Neither the architecture nor the universal weak method described here is unique, even given the problem-space hypothesis. Organizations other than the elaboration-decision-application cycle could be used for processing at a node. Much more needs to be learned about the computational issues, in order to understand the nature of acceptable architectures and universal weak methods.

The limitation on functionality plays a role in defining the architecture. The central concern is that problem solving on the basic functions of search control, given originally in Section 2, must not be limited by some fixed processing. Otherwise the intelligent behavior of the system would be inherently limited by these primitive acts. Subgoals are the general mechanism for bringing to bear the full intellectual resources of the agent. Thus, the appropriate solution (and the one taken here, although not worked out in this paper) is universal subgoaling, which shifts the processing limitation to the decision to evoke subgoaling.

The temptation is strong to ignore the limit on functionality and locate the intelligence within the processing at a node, and this approach has generated an entire line of problem-solving organizations. These usually focus on an *agenda* mechanism for what to do next, where the items on the agenda in effect determine both the state and the operator (although the organization is often not described in terms of problem spaces) (Erman, Hayes-Roth, Lesser, Reddy, 1980; Lenat, 1976; Nilsson, 1980). A particularly explicit version is the notion of *metarules* (Davis, 1980), which are rules to be used in deciding what rules are to be applied at the object-level, with of course the implied upward recursion to metametarules, etc. The motivation for such an organization is the same concern as expressed above, namely, that a given locus of selection (operators, states, etc.) be made intelligently. The two approaches can be contrasted by the one (metarules) providing a structurally distinct organization for such selections versus the other (SOAR) merging all levels into a single one, with subgoaling to permit the metadecisions to be recast at the object level.

That search control in SOAR operates with limited processing does not mean that the knowledge it brings to bear is small. In fact, the design problem for search control is precisely to reconcile maximizing the knowledge brought to bear with minimizing processing. This leads to casting search control as a *recognitional* architecture, that is, one that can recognize in the present state the appropriate facts or considerations and do so immediately, without extended inferential processing.[17] The use of a production system for search control

[17] The concept of recognition is not, of course, completely well defined, but it does convey the right flavor.

follows upon this view. Search control is indefinitely extendible in numbers of elaboration and decision productions. The time to select the relevant productions can be essentially independent of the number of productions, providing that the comparisons between the production conditions and the working memory elements are appropriately limited computationally.[18] This leads to the notion of a learning process that continually converts knowledge held in other ways into search-control productions. This is analogous to a mechanism of practice (Anderson, Greeno, Kline, Neves, 1981; Newell & Rosenbloom, 1981; Rosenbloom & Newell, 1982) and the modular characteristics of production systems make it possible. These are familiar properties of production systems; we mention them here because they enter into the computational characterization of search control.

The two phases of search control, elaboration and decision (voting), perform distinct functions and hence cannot be merged totally. Elaboration converts stored knowledge (in search control) and symbolized knowledge (in objects) into symbolized knowledge (in the current object). Voting converts stored knowledge and symbolized knowledge into an action (replacement of a current object). However, it is clearly possible to decrease the use of voting until it is a mere recognition of conventional signals associated with the objects, such as **select-me** and **reject-me**. In the other direction, shifting voting to respond directly to relevant task structure is equivalent to short circuiting the need to make all decisions explicit, with a possible increase in efficiency.

With limits on the computational power of search control, a universal weak method cannot realize all methods. It cannot even realize all versions of a given weak method. That is, there are varieties of a weak method, say hill climbing, that it can realize and varieties it cannot. It will be able to carry out the logic of the method, but not the computations required for the decisions. Thus, the correct claim for a universal weak method is that it provides for sufficiently computationally simple versions of all the weak methods, not that it can provide for all versions.

Imagine a space of all methods. There will be a distinguished null point, the default method, where no knowledge is available except that the task itself is represented. Then a universal weak method claims to realize a neighborhood of methods around this null point. These are methods that are sufficiently elementary in terms of the knowledge they demand about the task environment and the way they use this knowledge so that the knowledge can be directly converted into the appropriate action within the method. Other methods will be further away from the default point in the amount of knowledge incorporated and the inferences required to determine the method from that knowledge. To create the method from such knowledge requires

---

[18] Time per cycle can be logarithmic in the number of productions (providing the conditions are still limited), where the units of computation are the elementary comparisons (Forgy, 1982).

techniques analogous to program design and synthesis. Only inside some boundary around the default method will a universal weak method be able to provide the requisite method behavior.

Different universal weak methods will have boundaries at different places in the space of methods, depending on how much computational power is embodied in the processing power is in the architecture and how much knowledge is embodied in its search control. An adequate characterization of this boundary must wait until universal subgoaling is added to the universal weak method. But it is clear that the universal weak method is not unique.

6.5. Weak methods from knowledge of the task environment

The weak methods were factored into the universal weak method plus increments for each weak method ($S_U + S_M$). The increments were encodings of the particular knowledge of the environment that underlay the weak methods. One issue not dealt with is the conversion of task knowledge into the elaboration and decision productions of a method increment. This would have required a scheme for representing task knowledge independent of the agent to permit the conversion to be analyzed. We did provide two steps towards satisfying the condition that weak-method increments be easily derivable from task-environment knowledge. First, we required that a weak-method increment encode only the special knowledge used by the weak method. Thus, obtaining an increment became a local process. Second, the method increments themselves decomposed into concepts (computed by elaboration productions) and conversions of concepts to the appropriate action for the method (computed by the decision productions).

Only a few basic concepts occur in the collection of weak methods of Figure 5-3, beyond acceptability and failure, which occur in the UWM itself. Some are defined on the search behavior of the agent and are task independent: ancestor, current, depth, descendant, previous and produce. The rest depend on the specific task situation: can reduce, difference, duplication, estimated distance to the goal, evaluation and inverse. Other concepts would gradually be added as the number of methods increased. More important than the small number of these concepts is their extreme generality, which captures the fact that only notions that are applicable to almost any task are used in the weak methods.

If weak methods are generated by knowledge of the task environment, then weak methods should exist for each different state of knowledge, although conceivably an additional bit of task knowledge might not help. Also, this correspondence of knowledge to methods can only be expected within the computationally feasible region for the universal weak method. Although, as noted, we cannot explore this issue directly without the independent definition of a space of task knowledge, we can explore the issue at the level at which we do have a represention, namely, at the level of productions and their composition into method increments.

One question is whether simply combining the productions of existing method increments produces viable new methods or some sort of degenerate behavior. If the productions from Figure 5-3 are combined with each other, it is possible that productions will interfere with each other, causing the system to thrash among a small number of operators and states without exploring the problem space. Note, to begin with, that the methods in Figure 5-3 fall into two distinct types, either selecting states or selecting operators. Thus, we can break the analysis up into cases.

Combining together the S_M of a state-selection method with the S_M of an operator-selection method will not cause any interference, but only enhance the search (assuming that both methods are appropriate). This corresponds to improving a state-selection method by further narrowing the operator choices.

When operator-selection methods are combined, the productions will never cause the system to cease searching. It is possible that one method will vote for an operator and another vote against it, wiping each other out; but an operator will still be selected, specifically the one with the most total votes without any vetoes. Following the operator selection, the operator will be immediately applied to create a new state and cause the current operator to become undefined.

When state-selection methods are combined, the effects take two steps to work out. When state-selection methods are selecting a state, they combine in the same manner as operator selection methods. The votes are merely totalled and one state will win and become the current state. After the state is selected, the next decision phase would normally select an operator, however, the state could change again. This second chance at state selection opens the possibility for the reselection of the prior state, setting up an infinite loop. For instance, productions need not be independent of the current state, i.e., they will not vote for a state independent of whether it, or another state, is the current state. In steepest ascent hill climbing, the following production will vote for the ancestor of the current state, but will not vote for that state once it is the current state.

State: If the ancestor state is acceptable, vote for the ancestor state.

If this production alone was combined with a best-first method, an infinite loop would develop starting with the creation of a new best state by operator application. The above production would vote for the ancestor state, while the best-first method would vote for the current state. The tie would be broken by selecting the oldest state, at which point the above rule would not apply, but the best-first rule would vote in the best state, and the cycle would continue. This looping depends on the breadth-first nature of the architecture, but analogous loops can develop with other methods of tie-breaking. This problem does not appear for the methods we have described, because they all have the property that there is the same number of votes for a state independent of whether it is the current state. This is achieved in steepest ascent hill climbing by adding a production that votes for the current state if it is acceptable and its ancestor state is not acceptable. The only

general solution is to require new productions to be added in sets that obey the above property.

Instead of combining all of the productions from one method with another, it should be possible to generate new methods by an appropriate set of productions using the same concepts. Four new methods were in fact created during the investigation. The S_M productions for these methods appear in Figure 6-2. Each of these new methods consists of knowledge from previous methods, combined in new ways. All of them where implemented for at least one of the tasks.

The first new method, depth-first/breadth-second search (DFBrSS), has a single rule, which states that if the current state is acceptable, it should be voted for. As long as this production is true, the search will be depth first. The current state will remain selected, an operator will be selected, and then applied to create a new current state. This new state will then stay as the current state and the process will continue until an unacceptable state is encountered. At that point, the unacceptable state will be vetoed, and all other states will receive one vote. The tie will be broken by the architecture selecting the oldest acceptable state.[19] This gives the breadth-second character of the search. Following the selection by breaking the tie, the search will continue depth-first until another unacceptable state is encountered. Although this method relies on a feature of the architecture, we could add in the knowledge for breadth-first search, modified to apply when the current state is unacceptable, and achieve the same result.

The second new method is actually the implementation of simple hill climbing given earlier. It is included here because it differs from the classical implementation of simple hill climbing in that it falls back on the UWM, which produces a breadth-first search when a local maximum is reached. This can realize classical simple hill climbing by adding a production that detects the local maximum and returns the current state as the desired state. None of the tasks in our demonstration were simple maximization problems, so such a production was never needed.

The third and fourth methods are variations of simple hill climbing that deal with a local maximum in different ways. The first will select a descendant of the maximum from which to continue, giving a depth-second search. The second will select the best state other than the maximum, giving a best-second search.

[19] This was a "feature" of the XAPS2 architecture.

Depth-First/Breadth-Second Search (DFBrSS) [Used for Tower of Hanoi]
State: If the current state is acceptable, vote for it.
Simple Hill Climbing/Breadth-Second Search (SHCBrSS) [Used for Eight Puzzle, Missionaries, Picnic I and II]
State: If the current state is not acceptable or not as good as the ancestor state, vote for the ancestor state.
State: If the current state is acceptable and better than the ancestor state, vote for the current state.
Simple Hill Climbing/Depth-Second Search (SHCDSS) [Used for Eight Puzzle]
State: If the current state is not as good as the ancestor state, vote for the ancestor state.
State: If the current state is better than the ancestor state, vote for the current state.
State: If the current state is unacceptable, vote for its descendents.
State: If the current state is acceptable, and the ancestor state is unacceptable, vote for the current state.
Simple Hill Climbing/Best-Second Search (SHCBSS) [Used for Eight Puzzle]
State: If the current state is not as good as the ancestor state, vote for the ancestor state.
State: If the current state is better than the ancestor state, vote for the current state.
State: If the current state is unacceptable, vote for the best of its descendents.
State: If the current state is unacceptable, and there is only one descendant, vote for it.
State: If the current state is acceptable, and the ancestor state is unacceptable, vote for the current state.

Figure 6-2: Additional weak methods

6.6. Defining the weak methods

The universal weak method suggests a way to define the weak methods:

> A weak method is an AI method that is realized by a universal weak method plus some knowledge about the task environment (or the behavior of the agent).

This definition satisfies the characterization given at the beginning of Section 1. First, it makes only limited demands on the knowledge of the task environment. The proposed definition starts at the limited-knowledge end of the spectrum and proceeds toward methods that require more knowledge. At some point, as discussed, the form of automatic assimilation required by a universal weak method fails as the knowledge about the task environment becomes sufficiently complex. We do not know where such a boundary lies, but it would seem plausible to take as sufficiently limited any knowledge that could be immediately assimilated. Second, it provides a framework within which domain knowledge can be used. The amount of domain specific knowledge embedded in the concepts that enters into a weak method is limited in the first instance by the computational limits of search control. As we saw in discussing these limits, there can be an indefinite amount of recognitional knowledge. In the second instance subgoaling permits still more elaborate computations and the use of further knowledge, providing the use of the results remain as stipulated by the search control of the method increment.

The definition implies that the set of weak methods is relative to the universal weak method. As the latter varies in its characteristics, so too presumably does the set of weak methods. The set is also relative to the knowledge framework that can be used to form the increments to the universal weak methods. Finally, the definition is in terms of the *specification* of behavior, not of the behavior itself. Thus a weak method (e.g., hill climbing) can be represented both by an increment to a universal weak method and by some other specification device, eg, a Pascal program. All three of these implications are relatively novel, so that it is not clear at this juncture whether they make this proposed definition of weak methods more or less attractive.

This proposed definition of weak methods must remain open until some additional parts of the total organization come into being, especially universal subgoaling. Only then can a sufficiently exhaustive set of weak methods be expressed within this architecture to provide a strong test of the proposed definition. Additional aspects must also be examined, for example, fleshing out the existing collection of weak methods in various directions, such as methods for acquiring knowledge and methods for handling the various subgoals generated by universal subgoaling.

6.7. Relation to other work on methods

The work here endeavors to provide a qualitative theory for AI methods. Thus, it does not make immediate contact with much of the recent work on methods, which has sought to apply the research paradigm of algorithmic complexity to AI methods (see Banerji (1982) for a recent review).

However, it is useful to understand our relation to the work of Nau, Kumar and Kanal (1982). They have described a general form of branch and bound that they claim covers many of the search methods in AI. At a sufficiently general level, the two efforts express the same domination of a search framework. In terms of the details of the formulation, the two research efforts are complementary at the moment. They are concerned with a logical coverage of all forms of given methods under instantiation by specifying certain general functions. The intent is to integrate the analysis of a large number of methods. We are concerned with how an agent (not an analyst) can have a common organization for all members of a class of methods, under certain constraints of simplicity and directness. Their algorithm claims universal coverage of all forms of a given method; ours is limited to the neighborhood around the default search behavior as described in Section 6.7. However, in the longer run it is clear that the two research efforts could grow to speak to identical research questions -- if their algorithm became the base for a general problem-solving agent or if our universal weak method plus universal subgoaling came to have pretensions of extensive coverage. Then a detailed comparison would be useful.

A just-published note by Ernst and Banerji (1983) on the distinction between strong and weak problem solvers is also relevant. They wish to associate the strength of a problem solver with the formulation of the problem space -- strong solvers from good (knowledge-intensive) and weak solvers from weak (knowledge-lean) formulations. Once the formulation (the problem space) is given, then the problem solver is just an interpreter that runs off the behavior. This view agrees only in part with the one presented in this paper. Certainly the amount of knowledge available governs the strength of a problem solver -- weak methods use little knowledge. Certainly, also, the entire problem solving system is usefully viewed as an interpreter of a behavior specification -- the *[S, I]* of Section 1.1. Finally, the total knowledge involved in a problem is distributed amongst several components: the data structures that define the states, the operators, and the search control. The view of Ernst and Banerji seems to ignore the factorization between the problem space

(state representation and operators) and the search control, treating the latter as an unimportant contributor to the success of problem solving. The theory presented here takes the opposite view -- that after the space is given, heuristic knowledge must still be applied to obtain successful problem solving. Thus if this state is just an interpreter, as Ernst and Banerji maintain, it is nevertheless an interpreter that must still solve problems.

7. Conclusion

We have attempted in this paper to take a step towards a theory of AI methods and an appropriate architecture for a general intelligent agent. We introduced a specific problem-solving architecture, SOAR, based on the problem-space hypothesis, which treats all behavior as search and provides a form of behavior specification that factors the control into a recognition-like scheme (the elaboration-decision process) separate from the operators that perform the significant computations (steps in the problem space). Non-search behavior arises by the control being adequate to specify the correct operator at each step. This can be viewed as simply another programming formalism, which makes different assumptions about the default situation -- problematical rather than certain, as in standard languages.

On top of this architecture we introduced a universal weak method that provides the ability to perform as any weak method, given appropriate search-control increments that respond only to the special knowledge of the task environment. The existence of a universal weak method has implications for an agent being able to use the weak method appropriate to whatever knowledge it has about the task environment, without separate development of selection mechanisms that link knowledge of the task environment to methods. It also has implications for how weak methods are acquired, since it becomes a matter of acquiring the right elementary concepts with which to describe the environment, and does not require learning each weak method as a separate control structure.

Additional major steps are required to complete the theory. The most notable, and immediate is universal subgoaling. Entailed therein, in ways not yet completely worked out, is the need for processes of problem space and goal-state creation, since every subgoal must lead to a problem space and description of the goal states in order to provide actual solutions. But there are other steps as well. One is driving the factorization of weak methods back to the descriptive knowledge of the task environment, rather than just the productions of Figure 5-3, which combine descriptive and normative knowledge. A second is including the processes of planning, namely the construction of the plan as well as its implementation interpretively. A third is studying how the scheme behaves under the conditions of large bodies of search-control knowledge, rather than the lean search-controls that have been our emphasis here.

References

Amarel, S. An approach to heuristic problem-solving and theorem-proving in the propositional calculus. In Hart, J. & Takasu, S. (Eds.), *Systems and Computer Science.* Toronto: University of Toronto Press, 1967.

Anderson, J. R., Greeno, J. G., Kline, P. J. & Neves, D. M. Acquisition of problem solving skill. In Anderson, J. R. (Ed.), *Cognitive Skills and their Acquisition.* Hillsdale, NJ: Erlbaum, 1981.

Banerji, R. B. Theory of problem solving. *Proceedings of the IEEE*, 1982, *70*, 1428-1448.

Davis, R. Meta-rules: Reasoning about control. *Artificial Intelligence*, 1980, *15*, 179-222.

Duda, R. O. & Shortliffe, E. H. Expert systems research. *Science*, 1983, *220*, 261-268.

Erman, L., Hayes-Roth, F., Lesser, V., & Reddy, D. R. The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty. *Computing Surveys*, June 1980, *12*, 213-253.

Ernst, G. W. & Banerji, R. B. On the relationship between strong and weak problem solvers. *The AI Magazine*, 1983, *4*(2), 25-27.

Feigenbaum, E. A. & Feldman, J. (Eds.). *Computers and Thought.* New York: McGraw-Hill, 1963.

Fikes, R. E., Hart, P. E. & Nilsson, N. J. Learning and executing generalized robot plans. *Artificial Intelligence*, 1972, *3*(4), 251-288.

Forgy, C. L. *OPS5 Manual.* Computer Science Department, Carnegie-Mellon University, 1981.

Forgy, C. L. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 1982, *19*, 17-37.

Hayes, J. R. & Simon, H. A. Understanding complex task instructions. In Klahr, D. (Ed.), *Cognition and Instruction.* Hillsdale, NJ: Erlbaum, 1976.

Hewitt, C. *Description and Theoretical Analysis (using Schemata) of PLANNER: A language for proving theorems and manipulating models in a robot.* Doctoral dissertation, MIT, 1971.

Kowalski, R. *Logic for Problem Solving.* New York: North-Holland, 1979.

Laird, J. *Explorations of a Universal Weak Method* (Tech. Rep.). Computer Science Department, Carnegie-Mellon University, 1983. (In preparation).

Laird, J. & Newell, A. A universal weak method: Summary of results. In *Proceedings of the IJCAI-83.* Los Altos, CA: Kaufman, 1983.

Lenat, D. *AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search.* Doctoral dissertation, Computer Science Department, Stanford University, 1976.

Nau, D. S., Kumar, V. & Kanal, L. A general paradigm for A.I. search procedures. In *Proceedings of the AAAI82.* American Association for Artificial Intelligence, 1982.

Newell, A. Heuristic programming: Ill-structured problems. In Aronofsky, J. (Ed.), *Progress in Operations Research, III.* New York: Wiley, 1969.

Newell, A. Physical symbol systems. *Cognitive Science*, 1980, *4*, 135-183.

Newell, A. Reasoning, problem solving and decision processes: The problem space as a fundamental category. In R. Nickerson (Ed.), *Attention and Performance VIII.* Hillsdale, NJ: Erlbaum, 1980.

Newell, A. & Rosenbloom, P. Mechanisms of skill acquisition and the law of practice. In Anderson, J. A. (Ed.), *Learning and Cognition.* Hillsdale, NJ: Erlbaum, 1981.

Newell, A. & Simon. H. A. GPS, a program that simulates human thought. In Feigenbaum, E. A. & Feldman, J. (Eds.), *Computers and Thought.* New York: McGraw-Hill, 1963.

Newell, A. & Simon, H. A. *Human Problem Solving.* Englewood Cliffs: Prentice-Hall, 1972.

Newell, A. & Simon, H. A. Computer science as empirical inquiry: Symbols and search. *Communications of the ACM*, 1976, *19*(3), 113-126.

Newell, A., McDermott, J. & Forgy, C. L. *Artifical Intelligence: A self-paced introductory course* (Tech. Rep.). Computer Science Department, Carnegie-Mellon University, September 1977.

Nilsson, N. *Problem-solving Methods in Artificial Intelligence.* New York: McGraw-Hill, 1971.

Nilsson, N. *Principles of Artificial Intelligence.* Palo Alto, CA: Tioga, 1980.

Polya, G. *How to Solve It.* Princeton, NJ: Princeton University Press, 1945.

Polya, G. *Mathematical Discovery, 2 vols..* New York: Wiley, 1962.

Rosenbloom, P. S. & Newell, A. *Learning by chunking: A production-system model of practice* (Tech. Rep.). Computer Science Department, Carnegie-Mellon University, Oct 1982.

Rulifson, J. F., Derksen, J. A. & Waldinger, R. J. *QA4: A procedural calculus for intuitive reasoning* (Tech. Rep.). Stanford Research Institute Artificial Intelligence Center, 1972.

Sacerdoti, E. D. *A Structure for Plans and Behavior.* New York: Elsevier, 1977.

Wason, P. C. & Johnson-Laird, P. N. *Psychology of Reasoning: Structure and content.* Cambridge, MA: Harvard, 1972.

Waterman, D. A. & Hayes-Roth, F., (Eds.). *Pattern Directed Inference Systems.* New York: Academic Press, 1978.

Winston, P. *Artificial Intelligence.* Reading, MA: Addison-Wesley, 1977.