PROBLEM SOLVING TACTICS

Technical Note 189

July 1979

By: Earl D. Sacerdoti
Associate Director
Artifical Intelligence Center

SRI Project 7910

Preparation of this paper was supported by the Defense Advanced Reseach Projects Agency under contract N00038-79-C-0118 with the Naval Electronic Systems Command. Barbara Grosz, Peter Hart, Nils Nilsson, and Don Walker suggested helpful presentation tactics.

ABSTRACT

This paper describes the basic strategies of automatic problem solving, and then focuses on a variety of tactics for improving their efficiency. An attempt is made to provide some perspective on and structure to the set of tactics. Finally, some new directions for problem-solving research are discussed, and a personal perspective is provided on where the work is headed: toward greater flexibility of control and more intimate integration of plan generation, execution, and repair.

1. AUTOMATIC PROBLEM SOLVING*

For intelligent computers to be able to interact with the real world, they must be able to aggregate individual actions into sequences to achieve desired goals. This process is referred to as automatic problem solving, sometimes more casually called automatic planning. The sequences of actions that are generated are called plans.

Early work in automatic problem solving focused on what Newell [1] has called "weak methods." While these problem-solving strategies are quite general and are formally tractable, they are insufficient in practice for solving problems of any significant complexity. During the last decade, a number of techniques have been developed for improving the efficiency of these strategies. Since these techniques operate within the context of the general strategies, they are termed here problem-solving tactics. The bulk of this paper consists of a description of the problem-solving strategies and a catalogue of tactics for improving their efficiency. This is followed by an attempt to provide some perspective on and structure to the set of tactics. Finally, some new directions in problem-solving research are discussed, and a personal perspective is provided on where the work is headed: toward greater flexibility of control and more intimate integration of plan generation, execution, and repair.

Because problem solving involves exploration of alternative hypothesized sequences of actions, a symbolic model of the real world, referred to as a world model, is used to enable simple simulations of the critical aspects of the situation to be run as the plans are evolved. As with all models, the world models used in problem solving are abstractions or oversimplifications of the world they model.

---

* Preparation of this paper was supported by the Defense Advanced Research Projects Agency under contract N00039-79-C-0118 with the Naval Electronic Systems Command. Barbara Grosz, Peter Hart, Nils Nilsson, and Don Walker suggested helpful presentation tactics.

1.1. What is Needed to Generate Plans

The general function of an automatic problem solving system, then, is to construct a sequence of actions that transforms one world model into another. There are three basic capabilities that a problem solving system must have. These are:

a. Management of State Description Models - A *state description model* is a specification of the state of the world at some time. The facts or relations that are true at any particular time can be represented as some equivalent of predicate calculus formulas. (We shall refer, somewhat loosely, to these facts and relations as *attributes* of a state.) The critical aspect of representation for problem solving is the need to represent alternative and hypothetical situations, that is, to characterize the aggregate effects of alternative sequences of actions as the problem solver searches for a solution.

Three methods have typically been used for representing these alternatives. One method has been to include an explicit state specification in each literal or assertion (as suggested by McCarthy and Hayes [2] and implemented by Green [3]). Another alternative is to associate each literal with an implicit data context that can be explicitly referenced (as in QA4 [4]). A third choice is to have all the literals that describe the states explicitly tied up in the control structure of the problem solver (as, for example, in most problem solvers written in CONNIVER [5]).

b. Deductive Machinery - A state description model, then, contains all the information needed to characterize a particular state of the world. The information will not all be explicitly encoded, however, so a deductive engine of some sort must be provided to allow needed information to be extracted from a model. The deductions are of two types: within a particular state (this is where traditional, "monotonic" deduction systems are used), and across states (that is, reasoning about the effects of prior actions in a sequence). The deductive machinery can be viewed as a question-answering system that allows the problem solver to retrieve information about a particular state of the world from the state description model.

c. Action Models - In addition to state description models and a means of querying them, a problem solver must have a way of modelling what changes when an action is applied in an arbitrary state. Thus, an action is described by a mapping from one state description to another. Such a mapping is usually referred to as an operator. The mapping may be specified either by procedures, as in the problem solvers based on so-called AI languages [6], or by declarative data structures. In any case, they must specify at least the expressions that the action will make true in the world model and the expressions that its execution will make untrue in the world model. Usually, to help guide the heuristic search for actions that are relevant to achieve particular goals, one of the expressions to be made true by each operator is designated in some way as its "primary effect."

1.2. The Basic Control Strategy for Plan Generation

The process of generating a plan of action that achieves a desired goal state from a given initial state typically involves a extensive search among alternative sequences. A number of control strategies for tree search constitute the basic tools of all problem solving systems.

Problem solving systems usually work backward from the goal state to find a sequence of actions that could lead to it from the initial state. This procedure generates a tree of action sequences, with the goal state at the root, instances of operators defining the branches, and intermediate states defining the nodes. A tree search process of some sort is used to find a path to a node that corresponds to the initial state. The path from initial state to goal then defines the plan. Two particular tree search strategies are discussed here since they are so commonly used.

The first of these is means-ends analysis, which was the central search algorithm used by GPS [7] and STRIPS [8]. This strategy works as follows. The "difference" between the initial and goal states is determined, and that instance of the particular operator that would most reduce the difference is chosen.

If this operator is applicable in the initial state, it is applied, creating a new intermediate state. If the goal is satisfied in the new state, the search is completed. Otherwise, the difference between the new state and the goal state is determined, an operator to most reduce the new difference is chosen, and the process continues.

If the chosen operator is not applicable, its preconditions are established as a new intermediate subgoal. An attempt is made, using the search strategy recursively, to find a sequence of operators to achieve the subgoal state. If this can be done, the chosen operator is now applicable and the search proceeds as described above. If the new subgoal cannot be achieved, a new instance of an operator to reduce the difference is chosen and the process continues as before.

A second important search strategy, used in simple problem solvers written in the so-called AI languages [6], is backtracking, which works in the following manner. If the goal is satisfied in the initial state, a trivial solution has been found. If not, an operator that, if applied, would achieve the goal is selected. If it is applicable in the initial state, it is applied and a solution has been found. If the chosen operator is not applicable, operators that would achieve its preconditions are found, and the search proceeds as before to find plans to render them applicable. If the search fails, a different candidate operator is chosen and the process repeats.

This strategy follows a line of action out fully before rejecting it. It thus permits the search tree to be represented elegantly; all the active parts of the search tree can be encoded by the control stack of the search procedure itself, and all the inactive parts of the search tree need not be encoded at all. Because of the full search at each cycle of the process, it is critical that the correct operator be chosen first almost always. Otherwise, the simplicity of representation offered by this strategy will be amply repaid by the inefficiency of the search.

As was discussed above, these strategies are insufficient in practice for solving problems of any significant complexity. In particular, one of the most costly behaviors of the basic problem solving strategies is their inefficiency in dealing with goal descriptions that include conjunctions. Because there is usually no good reason for the problem solver to prefer to attack one conjunct before another, an incorrect ordering will often be chosen. This can lead to an extensive search for a sequence of actions to try to achieve subgoals in an unachievable order.

2. TACTICS FOR EFFICIENT PROBLEM SOLVING

2.1. Hierarchical Planning

The general strategies described above apply a uniform procedure to the action descriptions and state descriptions that they are given. Thus, they have no inherent ability to distinguish what is important from what is a detail. However, some aspects of almost any problem are significantly more important than others. By employing additional knowledge about the ranking in importance of aspects of the problem description, a problem solver can concentrate its efforts on the decisions that are critical while spending less effort on those that are relatively unimportant.

Information about importance can be used in several ways. First, the standard strategies can be modified to deal with the most important (and most difficult to achieve) subgoals first. The solution to the most important subgoals often leaves the world model in a state from which the less important subgoals are still achievable (if not, the weaker search strategies must be employed as a last resort). The less important subgoals could presumably be solved in many ways, some of which would be incompatible with the eventual solution to the important subgoals. Thus, this approach constrains the search where the constraints are important, and avoids overconstraining it by making premature choices about how to solve less important aspects of the problem. Siklossy and Dreussi [9] used an explicit ordering of types of subgoals to guide search.

Another way to focus on the critical decisions first is to abstract the descriptions of the actions, thereby creating a simpler problem. This abstracted problem can be solved, producing a sequence of abstracted actions, and this plan can then be used as a skeleton, identifying critical subgoals along the way to a solution, around which

to construct a fully detailed plan. This tactic was used in conjunction with an early version of GPS by Newell, Shaw, and Simon [7] to find proofs in symbolic logic (using abstracted operators and state descriptions that ignored the connectives and the ordering of symbols).

Finally, abstraction can be extended to involve multiple levels, leading to a hierarchy of plans, each serving as a skeleton for the problem solving process at the next level of detail. The search process at each level of detail can thus be reduced to a sequence of relatively simple subproblems of achieving the preconditions of the next step in the skeleton plan from an initial state in which the previous step in the skeleton plan has just been achieved. In this way, rather complex problems can be reduced to a sequence of much shorter, simpler subproblems. Sacerdoti applied this tactic to robot navigation tasks [10] and to more complex tasks involving assembly of machine components [11].

2.2. Hierarchical Plan Repair

A side-effect of hierarchical planning is that plans can possibly be created that appear to be workable at a high level of abstraction but whose detailed expansions turn out to be invalid. The basic idea behind the plan repair tactic is to check, as a higher level plan is being expanded, that all the intended effects of the sequence of higher-level actions are indeed being achieved by the collection of subsequences of lower-level actions. By exploiting the hierarchical structure of the plan, only a small number of effects need to be checked for. Various methods for patching up the failed plan can then be applied. This tactic was incorporated in a running system by Sacerdoti [11] and is very similar to a technique called "hierarchical debugging" articulated by Goldstein [12] for a program understanding task.

2.3. Bugging

Rather than attempt to produce perfect plans on the first attempt, it can often be more efficient to produce an initial plan that is approximately correct but contains some "bugs," and subsequently alter the plan to remove the bugs. By employing additional knowledge about bug classification and debugging, this approach allows the decisions made during the problem-solving process about which action to try next to be made with less effort, since mistaken decisions can be subsequently fixed.

Sussman, who first employed this tactic in his HACKER system [13], called it "problem-solving by debugging almost-right plans." It is often referred to in the literature as the "debugging approach" and, indeed, it has spawned interesting research in techniques for debugging programs or plans developed both by machines and by people (see, for example, Sussman [14] and Goldstein and Miller [15]). Debugging, however, is an integral part of the execution component of any problem solver. What distinguishes this approach is a tolerance for introducing bugs while generating the plan, and thus it can more accurately be called the "bugging" approach.

This tactic works by deliberately making assumptions that oversimplify the problem of integrating multiple subplans. These assumptions may cause the problem solver to produce an initial plan with bugs in it. However, if the oversimplifications are designed properly, then only bugs of a limited number of types will be introduced, and relatively simple mechanisms can be implemented to remedy each expected type of bug.

2.4. Special-Purpose Subplanners

Once a particular subgoal has been generated, it may well be the case that it is of a type for which a special purpose algorithm, a stronger method than the weak method of the general-purpose problem solver, can be brought to bear. For example, in a robot problem, the achievement of an INROOM goal can be performed by a route-finding

algorithm applied to a connectivity graph representing the interconnection of rooms, rather than using more general methods applied to less direct representations of the rooms in the environment. Such a special purpose problem solver was used by Siklossy and Dreussi [9] to effect dramatic improvements in the system's performance.

Wilkins [16] employs special-purpose subplanners in a chess problem solver for subgoals such as moving safely to a given square or checking with a given piece.

Each special-purpose subplanner encodes additional knowledge about its specialty. To take advantage of it, the problem solver must incorporate information about how to recognize the special situation as well.

2.5. Constraint Satisfaction

Constraint satisfaction, the derivation of globally consistent assignments of values to variables subject to local constraints, is not usually thought of as a problem solving tactic. While it cannot be used to generate action sequences, it can play a very important role in particular subproblems, especially in determining the binding of variables when there is no clear locally computable reason to prefer one value over another. From this perspective, constraint satisfaction can be thought of as a type of special-purpose subplanner.

Stefik [17] employs constraint satisfaction to assign values to variables in generating action sequences for molecular genetics experiments.

2.6. Relevant Backtracking

As a correct plan is being searched for, a problem solver will encounter many choice points at which there are several alternative steps to be taken. Most problem solvers employ sophisticated techniques to try to make the right choices among the alternative action sequences initially. Alternatively, a problem solver could focus on sophisticated

post-mortem analyses of the information gained from early attempts that fail. By analyzing the reasons for the failure of a particular sequence of actions, the problem solver can determine which action in the sequence should be modified. This is in contrast with the straightforward approach of backtracking to the most recent choice point and trying other alternatives there. Fahlman [18] developed such a system for planning the construction of complex block structures. His system associated a "gripe handler" with each choice point as it was encountered, and developed a characterization of each failure when it occurred. When a particular line of action failed, the gripe handlers would be invoked in reverse chronological order to see if they could suggest something specific to do about the failure. The effect of this mechanism is to backtrack not to the most recent choice point, but to the relevant choice point.

The tactic of relevant backtracking, which is also referred to in the literature as dependency-directed or non-chronological backtracking, was also used in a problem solver for computer-aided design developed by Latombe [19].

2.7. Disproving

Problem solvers have traditionally been automated optimists. They presume that a solution to each problem or subproblem can be found if only the right combination of primitive actions can be put together. Thus, the impossibility of achieving a given goal or subgoal state can only be discovered after an exhaustive search of all the possible combinations of potentially relevant actions.

It may well be the case that a pessimistic analysis of a particular goal, developing knowledge additional to that employed in building action sequences, would quickly show the futility of the whole endeavor. This procedure can be of particular value in evaluating a set of conjunctive subgoals to avoid working on any of them when one can be shown to be impossible. Furthermore, even if a goal cannot be shown to be impossible, the additional knowledge might suggest an action sequence

that would achieve the goal. Siklossy and Roach [20] developed a system that integrated attempts to achieve goals with attempts to prove their impossibility.

2.8. Pseudo-Reduction

One of the most costly behaviors of problem solving systems is their inefficiency in dealing with goal descriptions that include conjunctions, as was noted at the end of Section 1. Ordering the conjuncts by importance, as described in the subsection on hierarchical planning above, can help, but there may still be multiple conjuncts of the same importance. One approach to the problem of selecting an order for the conjuncts is to ignore ordering them initially, finding a plan to achieve each conjunct independently. Thus, the conjunctive problem is reduced to several simpler, nonconjunctive problems. Of course, the plans to solve the reduced problems must then be integrated, using knowledge about how plan segments can be intertwined without destroying their important effects.

This tactic creates plans that are not linear orderings of actions with respect to time, but are rather partial orderings. This renders the cross-state question-answering procedure described in Section 1.1 more complicated than for other tactics.

By avoiding premature commitments to particular orderings of subgoals, this tactic eliminates much of the backtracking typical of problem solving systems. Pseudo-reduction was developed by Sacerdoti [11] and has been applied to robot problems, assembly of electromechanical equipment, and project planning [21].

2.9. Goal Regression

The problem-solving tactics we have discussed so far all work by modifying, in one way or another, the sequence of actions being developed to satisfy the goals. Goal regression modifies the goals as well. It relies on the fact that, given a particular goal and a particular action, it is possible to derive a new goal such that if the

new goal is true before the action is executed, then the original goal will be true after the action is executed. The computation of the new goal, given the original goal and the action, is called regressing the goal over the action.

As an example, let us suppose the overall goal consists of two conjunctive subgoals. This tactic first tries to achieve the second goal in a context in which a sequence of actions has achieved the first goal (similarly to the bugging tactic).

If this fails, the second goal is regressed back across the last action that achieved the first goal. This process generates a new goal that describes a state such that if the last action were executed in it, would lead to a state in which the original second goal were true. If the regressed goal can be achieved without destroying the first goal, the tactic has succeeded. If not, the regression process continues.

This tactic requires knowledge of the inverse effects of each operator. That is, in addition to knowing how the subsequent application of an operator changes a world model, the system must know how the prior application of the operator affects a goal.

The goal regression technique was developed independently by Waldinger [22] and Warren [23].

3. WHAT'S GOING ON?

We have just finished a brief (heuristically) guided tour of some of the problem-solving tactics used recently. They constitute a diverse bag of tricks for improving the efficiency of the problem solving process. In this section we focus on the underlying reasons why these techniques seem to help.

Problem-solving is often described as state-space search, or as exploration of a tree of possible action sequences. We can find some structure for the bag of tactical tricks by remembering that search or exploration involves not only movement to new (conceptual) locations, but discovery and learning as well.

The problem solver begins its work with information about only the initial state and the goal state. It must acquire information about the intermediate states as it explores them. It must summarize this information for efficient use during the rest of the problem-solving process, and it must take advantage of all possible information that can be extracted from each intermediate state. This can require considerable computational resources. In the simplest search strategies, the information may be simply a number representing the value of the heuristic evaluation function applied at that point. It may be much more, however. It may include a detailed data base describing the situation, information about how to deal with classes of anticipated subsequent errors, and dependency relationships among the attributes describing the situation. All this information is typically stored in intermediate contexts in one of the forms discussed in the first section of this paper.

The information learned during the exploration process can be broken down into four kinds of relationships among the actions in a plan. These are:

order relationships - the sequencing of the actions in the plan;

hierarchical relationships - the links between each action at one level and the meta-actions above it and the more detailed actions below it;

teleological relationships - the purposes for which each action has been placed in the plan; and

object relationships - the dependencies among the objects that are being manipulated (which correspond to dependencies among the parameters or variables in the operators).

These relationships can be explicated and understood only by carrying out the instantiation of new points in the search space. It is thus of high value to a problem solver to instantiate and learn about new intermediate states. As a simple example of a problem-solving tactic that displays incremental learning, consider relevant backtracking. By following initial paths through the search tree, the problem solver learns which choice points are critical. Another clear example is constraint satisfaction, in which restrictions on acceptable bindings for variables are aggregated as search progresses.

The difficulty is that, as with all learning systems, the acquisition of new information is expensive. The generation of each new state is a major time consumer in many problem solving systems. Furthermore, the generation of each intermediate state represents a commitment to a particular line of action by the problem solver. Since problems of any non-toy level of complexity tend to generate very bushy search trees, a breadth-first search strategy is impossible to use. Therefore, once a line of action has begun to be investigated, a problem-solving system will tend to continue with it. It is thus of high value to a problem solver, whenever possible, to avoid generating intermediate states not on the solution path. Those states that are generated must represent a good investment for the problem solver.

Thus, there are two opposing ways to improve the efficiency of a problem solver. The first is to employ a relatively expensive evaluation function and to work hard to avoid generating states not on the eventual solution path. The second is to use a cheap evaluation

function, to explore lots of paths that might not work out, but to acquire information about the interrelationships of the actions and objects in the world in the process. This information can then be used to guide (efficiently) subsequent search.

Each of the tactics described in the previous section strikes a particular balance between the value of instantiating new intermediate states and the cost of commitment to particular lines of action. While none of the tactics use one approach exclusively, each can be categorized by the one it emphasizes. Furthermore, each can be distinguished according to one of the four types of relationships they depend on or exploit. Table 1 displays a candidate categorization.

Relationship:	Approach: Learn and Summarize	Choose New Move
Order	pseudo-red. generation relevant backtracking disproving	
Hierarchy	plan repair	spec.-purpose subplanners
Teleology	bugging pseudo-red. critics	regression
Object	relevant backtracking	constraint satisfaction

TABLE 1

Classification of Tactics

None of the tactics fit as neatly into the classification as the table suggests, because they typically have been embodied in a complete problem solving system and so must deal with at least some aspects of many of the categories.

4. WHAT'S NEXT?

The current state of the art in plan generation allows for planning in a basically hierarchical fashion, using a severely limited (and predetermined) subset of the tactics enumerated above, by and for a single active agent satisfying a set of goals completely. The elimination of these restrictions is a challenge to workers in Artificial Intelligence. This section will discuss a number of these restrictions briefly and suggest, where possible, lines of research to ease them.

4.1. Integrating the Tactics

To date, there has been no successful attempt known to this author to integrate a significant number of the tactics we have described into a single system. What follows are some preliminary thoughts on how such an integration might be achieved.

First of all, the technique of hierarchical planning can be applied independently of any of the others. That is, all of the other techniques can be applied at each level of detail within the hierarchy. A number of interesting problems (analogous to that faced by the "hierarchical plan repair" tactic) would have to be faced in integrating their application across levels of the hierarchy.

Approaches for dealing with each of the types of relationships shown in Table 1 can probably be selected without major impact on the approaches selected for the other relationships. Thus, we can use Table 1 as a menu of possible tactics from which various collections can be constructed that make sense together in a problem solver.

Tactics that emphasize the clever selection of new paths to explore in the search space might be difficult to integrate with tactics that

emphasize the learning and summarization of information derived from the portion of the search space already explored. However, the major payoff in integrating tactics might come from exactly this kind of combination. Developing a problem solver that uses both kinds of technique when appropriate will probably require the use of novel control strategies. The interesting new results from such an endeavor will derive from efforts to employ information developed by one tactic in the application of other tactics.

4.2. Flexible Control Structure

While the tactic of hierarchical planning speeds up the problem solving process greatly, it requires that a plan be fully developed to the finest detail before it is executed. In real-world environments where unexpected events occur frequently and the detailed outcome of particular actions may vary, creation of a complete plan before execution is not appropriate. Rather, the plan should be roughed out and its critical segments created in detail. The critical segments will certainly include those that must be executed first, but also may include other aspects of the plan at conceptual "choke points" where the details of a subplan may affect grosser aspects of other parts of the plan.

Hayes-Roth *et al.* [24] are developing a program based on a model of problem solving that would produce the kind of non-top-down behavior suggested here. Their model is based on a Hearsay-II architecture [25], but could probably be implemented using any methodology that allowed for explicit analysis of each of the four kinds of dependencies described in Section III above. Stefik [17] has implemented a system that, at least in principle, has the power to produce this kind of behavior. His system incorporates a flexible means of determining which planning decision to make next. His decisions are local ones; should global ones be incorporated as well, we might see a means of determining dynamically which tactic to employ in a given situation.

4.3. Planning for Parallel Execution

Problem solvers to date have been written with the idea that the plan is to be generated by a single processor and will ultimately be executed one step at a time. The development of cooperating problem solvers and algorithms for execution by multiple effectors will force a closer look at the structure of plans and the nature of the interactions between actions.

A solid start has been made in this area. Fikes, Hart, and Nilsson [26] proposed an algorithm for partitioning a plan among multiple effectors. Smith [27] developed a problem solver that distributes both the plan generation and execution tasks. The pseudo-reduction tactic creates plans that are partially ordered with respect to time, and are therefore amenable both to planning in parallel by multiple problem solvers and to execution in parallel by multiple effectors. Corkill [28] is adapting the NOAH [11] pseudo-reduction problem solver to use multiple processors in plan generation, and we at SRI are adapting it to plan for the use of multiple effectors.

4.4. Partial Goal Fulfillment

Problem solvers to date have been designed to fully satisfy their goals. As the problems we work with become more complex, and as we attempt to integrate problem solvers with execution routines to control real-world behavior, full goal satisfaction will be impossible. In particular, a system that deals with the real world may need to execute a partially satisfactory plan and see how the world reacts to it before being able to complete the next increment of planning. Thus, we must be able to plan for the partial satisfaction of a set of goals. This implies that a means must be found of prioritizing the goals and of recognizing when an adequate increment in the planning process has been achieved.

5. A PERSPECTIVE

The problems we have posed have a common theme to their solution: increased flexibility in the planning process. The metaphor of problem solving as exploration for information that was presented above suggests that the result of pursuing the problem solving process can lead to surprises as great as those encountered in plan execution. The plan execution components of problem solving systems have been forced to be quite flexible because of the surprises from the real world that they had to deal with. Therefore, especially as the tactics used in exploring for plans become more daring, the lessons that can be learned from plan execution can be extremely valuable for plan generation.

This view suggests a direction for future work in problem solving: it will become more like incremental plan repair. The means of storing and querying state description models will have to allow for efficient updating when the orders of actions are altered and when new actions are inserted in mid-plan. Planning at higher levels of abstraction will appear very similar to planning for information acquisition during plan execution.

Therefore, the best research strategy for advancing the state of the art in problem solving might well be to focus on integrated systems for plan generation, execution, and repair. By developing catalogues of plan execution tactics and plan repair tactics to accompany this catalogue of plan generation tactics, we can begin to deal with problems drawn from rich, interactive environments that have thus far been beyond us.

REFERENCES

1. A. Newell, "Heuristic Programming: Ill-Structured Problems," in J. Aronofsky (ed.) Progress in Operations Research, Vol. III, pp. 360-414 (Wiley, New York, 1969).

2. J. McCarthy and P. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence," Machine Intelligence 4, B. Meltzer and D. Michie, eds., pp. 463-502 (American Elsevier, New York, 1969).

3. C. Green, "The Application of Theorem-Proving to Question-Answering Systems," Doctoral dissertation, Elec. Eng. Dept., Stanford Univ., Stanford, CA, June 1969. Also printed as Stanford Artificial Intelligence Project Memo AI-96 (June 1969).

4. R.F. Rulifson, J.A. Derksen and R. J. Waldinger "QA4: A Procedural Calculus for Intuitive Reasoning," Artificial Intelligence Center, Technical Note 73, Stanford Research Institute, Menlo Park, California (November 1972).

5. D.V. McDermott and G.J. Sussman, "The CONNIVER Reference Manual," MIT, Artificial Intelligence Lab., Memo No. 259, Cambridge, MA (May 1972).

6. D.G. Bobrow and B. Raphael, "New Programming Languages for Artificial Intelligence," Computing Surveys, Vol. 6, No. 3 (Sept. 1974).

7. G.W. Ernst and A. Newell GPS: A Case Study in Generality and Problem Solving (Academic Press, New York, 1969).

8. R.E. Fikes and N.J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," Artificial Intelligence, Vol. 2, No. 3-4, pp. 189-208 (Winter 1971).

9. L. Siklossy and J. Dreussi, "An Efficient Robot Planner Which Generates Its Own Procedures," Proc. Third International Joint Conference on Artificial Intelligence, Stanford, California, (August 1973).

10. E.D. Sacerdoti, "Planning in a Hierarchy of Abstraction Spaces," Artificial Intelligence, Vol. 5 No. 2, pp. 115-135 (Summer 1974).

11. E.D. Sacerdoti, A Structure for Plans and Behavior, (Elsevier North-Holland, New York, 1977).

12. I.P. Goldstein, "Understanding Simple Picture Programs," Tech. Note AI-TR-294, Artificial Intelligence Laboratory, MIT, Cambridge, MA (September 1974).

13. G.J. Sussman, A Computer Model of Skill Acquisition, (American Elsevier, New York, 1975).

14. G.J. Sussman, "The Virtuous Nature of Bugs," Proc. AISB Summer Conference, pp. 224-237 (July 1974).

15. M.L. Miller and I.P. Goldstein, "Structured Planning and Debugging," Proc. Fifth International Joint Conference on Artificial Intelligence, pp. 773-779, Cambridge, Massachusetts (August 1977).

16. D. Wilkins, "Using Plans in Chess," Proc. Sixth International Joint Conference on Artificial Intelligence, Tokyo, Japan (August 1979).

17. M.J. Stefik, "Orthogonal Planning with Constraints, Report on a Knowledge-Based Program that Plans Synthesis Experiments in Molecular Genetics," forthcoming dissertation, Computer Science Department, Stanford University (September 1979).

18. S.E. Fahlman, "A Planning System for Robot Construction Tasks," Artificial Intelligence, Vol. 5, No. 1, pp. 1-49 (Spring 1974).

19. J.C. Latombe, "Artificial Intelligence in Computer-Aided Design: the TROPIC System," in J.J. Allen (ed.), CAD Systems (Elsevier North-Holland, New York, 1977).

20. L. Siklossy and J. Roach, "Collaborative Problem-Solving between Optimistic and Pessimistic Problem Solvers," Proc. IFIP Congress 74, pp. 814-817 (North-Holland Publishing Company, 1974).

21. A. Tate, "Generating Project Networks," Proc. Fifth International Joint Conference on Artificial Intelligence, pp. 888-893, Cambridge, Massachusetts (August 1977).

22. R. Waldinger, "Achieving Several Goals Simultaneously," in E.W. Elcock and D. Michie (eds.) Machine Intelligence 8, pp. 94-136 (Ellis Horwood Limited, Chichester, England, 1977).

23. D.H.D. Warren, "WARPLAN: A System for Generating Plans," Department of Computational Logic, Memo No. 76, University of Edinburgh, Edinburgh (June 1974).

24. B. Hayes-Roth, F. Hayes-Roth, S. Rosenschein, and S. Cammarata, "Modeling Planning as an Incremental, Opportunistic Process," Proc. Sixth International Joint Conference on Artificial Intelligence, Tokyo, Japan (August 1979).

25. V.R. Lesser and L.D. Erman, "A Retrospective View of the Hearsay-II Architecture," Proc. Fifth International Joint Conference on Artificial Intelligence, pp. 790-800, Cambridge, Massachusetts (August 1977).

26. R.E. Fikes, P.E.Hart, and N.J. Nilsson, "Some New Directions in Robot Problem Solving," in B. Meltzer and D. Michie (eds.) Machine Intelligence 7, pp. 405-430 (Edinburgh Univ. Press, Edinburgh, 1972).

27. R.G. Smith, "A Framework for Distributed Problem Solving," Proc. Sixth International Joint Conference on Artificial Intelligence, Tokyo, Japan (August 1979).

28. D.D. Corkill, "Hierarchical Planning in a Distributed Environment," Proc. Sixth International Joint Conference on Artificial Intelligence, Tokyo, Japan (August 1979).