October 1982
Also numbered HPP-82-21

Report No. STAN-CS-82-939

① ok DTIC

85 - 0204

Planning and Problem Solving

by

Paul R. Cohen

Department of Computer Science

Stanford University
Stanford, CA 94305

LELAND STANFORD JUNIOR UNIVERSITY
ORGANIZED 1891

[redacted]

Abstract:

This report is reproduced from Chapter XV, "Planning and Problem Solving," of the *Handbook of Artificial Intelligence* (Vol. III, edited by Paul R. Cohen and Edward A. Feigenbaum). The chapter was written by Paul R. Cohen, with contributions by Stephen Westfold and Peter Friedland. Intended as an extension of Chapter II in Volume I on search, this chapter reviews nonhierarchical planning and continues on to discuss hierarchical and least-commitment planning and the refinement of skeletal plans.

Accesion For	
NTIS CRA&I	X
DTIC TAB	☐
Unannounced	☐
Justification	
By per call	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	20

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER STAN-CS-82-939; HPP-82-21	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Planning and Problem Solving		5. TYPE OF REPORT & PERIOD COVERED technical, July 1982
		6. PERFORMING ORG. REPORT NUMBER STAN-CS-82-939; HPP-82-21
7. AUTHOR(s) Paul R. Cohen (edited by Paul R. Cohen and Edward A. Feigenbaum)		8. CONTRACT OR GRANT NUMBER(s) MDA 903-80-C-0107
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science Stanford University Stanford, California 94305 U.S.A.		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency Information Processing Techniques Office 1400 Wilson Avenue, Arlington, VA 22209		12. REPORT DATE July 1982 13. NO. OF PAGES 61
14. MONITORING AGENCY NAME & ADDRESS (if diff. from Controlling Office) Mr. Robin Simpson, Resident Representative Office of Naval Research, Durand 165 Stanford University		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this report)

~~Reproduction in whole or in part is permitted for any purpose of the U.S. Government.~~

Distribution authorized to U. S. Government agencies and private individuals or enterprises eligible to obtain export controlled technical data in accordance with ~~implementing 10 USC 140c~~ Other requests must be referred to DARPA/TIO 1400 Wilson Bl. ~~Arlington, Va. 22209~~ (X)

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

DODD 5230.25 3/18/87

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report is reproduced from Chapter XV, "Planning and Problem Solving," of the Handbook of Artificial Intelligence (Vol. III, edited by Paul R. Cohen and Edward A. Feigenbaum). The chapter was written by Paul Cohen, with contributions by Steve Westfold and Peter Friedland. This chapter, intended as an extension of Chapter II in Volume I on search, reviews nonhierarchical planning and continues on to discuss hierarchical and least-commitment planning and the refinement of skeletal plans.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19. KEY WORDS (Continued)

20 ABSTRACT (Continued)

DD FORM 1 JAN 73 1473 (BACK)

EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Planning and Problem Solving

by

Paul R. Cohen

Chapter XV of Volume III of the

Handbook of Artificial Intelligence

edited by

Paul R. Cohen and Edward A. Feigenbaum

This research was supported by both the Defense Advanced Research Projects Agency (ARPA Order No. 3423, Contract No. MDA 903-80-C-0107) and the SUMEX-AIM Computer Project under the National Institutes of Health (Grant No. NIH RR-00785). The views and conclusions of this document should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the National Institutes of Health, or the United States Government.

CHAPTER XV: PLANNING AND PROBLEM SOLVING

FOREWORD

The Handbook of Artificial Intelligence was conceived in 1975 by Professor Edward A. Feigenbaum as a compendium of knowledge of AI and its applications. In the ensuing years, students and AI researchers at Stanford's Department of Computer Science, a major center for AI research, and at universities and laboratories across the nation have contributed to the project. The scope of the work is broad: About 200 short articles cover most of the important ideas, techniques, and systems developed during 25 years of research in AI.

Overview articles in each chapter describe the basic issues, alternative approaches, and unsolved problems that characterize areas of AI; they are the best critical discussions anywhere of activity in the field. These, as well as the more technical articles, are carefully edited to remove confusing and unessential jargon, key concepts are introduced with thorough explanations (usually in the overview articles), and the three volumes are completely indexed and cross-referenced to make it clear how the important ideas of AI relate to each other. Finally, the *Handbook* is organized hierarchically, so that readers can choose how deeply into the detail of each chapter they wish to penetrate.

This technical report is reproduced from Chapter XV, "Planning and Problem Solving," of the *Handbook* (Vol. III, edited by Paul R. Cohen and Edward A. Feigenbaum). The chapter was written by Paul R. Cohen. Stephen Westfold wrote an early version of the NOAH article, and Peter Friedland wrote the article on the refinement of skeletal plans. Intended as an extension of Chapter II in Volume I on search, this chapter reviews nonhierarchical planning and continues on to discuss hierarchical and least-commitment planning and the refinement of skeletal plans.

A

artificial intelligence

A. OVERVIEW

PROBLEM SOLVING is the process of developing a sequence of actions to achieve a goal. This broad definition admits all goal-directed ~~AI~~ programs to the ranks of problem solvers: for example, MYCIN ~~(see Article VIII.B1, in Vol. II)~~ solves the problem of determining a bacteremia infection. HARPY ~~(Article V.C2, in Vol. I)~~ solves the problem of understanding speech signals. and ~~AM (Article XIV.D1c)~~ solves the problem of filling in slots in its representations of concepts It follows that this chapter is not about problem solvers—the entire *Handbook* is about problem solvers. This ~~chapter~~, like the chapter on search (Chap. II, in Vol. I). is about problem-solving techniques. In particular, it is about *planning.*

In everyday terms, planning means deciding on a course of action before acting. This definition accurately describes the planning systems of this chapter. so we will adopt it. A plan is, thus, a representation of a course of action. It can be an unordered list of goals, such as a grocery list, but usually a plan has an implicit ordering of its goals; for example, most people plan to get dressed to go to the theater, not the other way around. Many plans include steps that are vague and require further specification. These serve as placeholders in a plan; for example, a daily plan includes the goal *eat-lunch,* although the details—where to eat, what to eat, when to leave—are not specified. The detailed plan associated with eating lunch is a *subplan* of the overall daily plan. Most plans have a rich subplan structure; each goal in a plan can be replaced by a more detailed subplan to achieve it. Although a finished plan is a *linear* or *partial* ordering of problem-solving operators, the goals achieved by the operators often have a hierarchical structure (see Fig. A-1). This aspect of plans prompted one of the earliest definitions:

> A Plan is any hierarchical process in the organism that can control the order in which a sequence of ~~operations~~ is to be performed. (Miller, Galanter, and Pribram, 1960, p. 16)

(KR)

Planning and Problem Solving

Failure to plan can result in less than optimal problem solving; one may go to the library twice, for example, having failed to plan to borrow a book and return another at the same time. Moreover, in cases where goals are not independent. failing to plan before acting may actually preclude a solution to the problem. For example, the goal of building a house includes the subgoals of installing a dry wall and installing electrical wiring, but these goals are not independent. The wiring must be installed first: otherwise, the dry wall will be in the way.

Plans can be used to monitor progress during problem solving and to catch errors before they do too much harm. This is especially important if the problem solver is not the only actor in the problem solver's environment and if the environment can change in unpredictable ways. Consider the example of a roving vehicle on a distant planet: It must be able to plan a route and then replan if it finds that the state of the world is not as it expected. *Feedback* about the state of the world is compared with what is predicted by the plan, which can then be modified in the event of discrepancies. This topic is discussed more fully in Sacerdoti (1975). The benefits of planning can be summarized as reducing search, resolving goal conflicts, and providing a basis for error recovery. These will be discussed in detail in the remainder of this chapter.

Approaches to Planning

Four distinct approaches to planning are discussed in this volume. They are nonhierarchical planning, hierarchical planning, script-based planning, and opportunistic planning. Here we must resolve a confusing ambiguity in the word *hierarchical.* The vast majority of plans have nested subgoal structures—*hierarchical structures*—as shown in Figure A-1. However, the word has another interpretation, one that provides the basis for distinguishing hierarchical from nonhierarchical planning. The distinction is that hierarchical planners generate a hierarchy of ***representations*** of a plan in which the highest is a simplification, or ***abstraction***, of the plan and the lowest

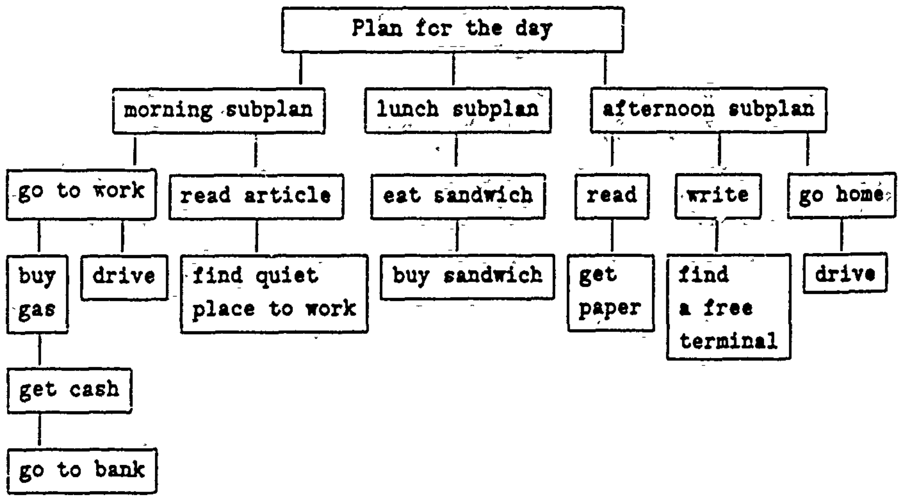


Figure A-1. Plan for a day, illustrating the hierarchical structure of subplans.

is a detailed plan, sufficient to solve the problem. In contrast, nonhierarchical planners have only one representation of a plan. Both kinds of planners generate plans with hierarchical subgoal structures, but only hierarchical planners utilize a hierarchy of representations of the plan. This distinction is discussed further in Article XV.B, in which STRIPS (a nonhierarchical planner) and ABSTRIPS (the hierarchical extension of STRIPS) are compared.

Nonhierarchical planning corresponds roughly to the colloquial meaning of planning; that is, a nonhierarchical planner develops a sequence of problem-solving actions to achieve each of its goals. It may *reduce* goals to simpler ones, or it may use *means-ends analysis* to reduce the *differences* between the current state of the world and that would hold after the problem has been solved. Examples of nonhierarchical planners are STRIPS (Article XV.B), HACKER (Article XV.C), and INTERPLAN (also in Article XV.C).

The major disadvantage of nonhierarchical planning is that it does not distinguish between problem-solving actions that are critical to the success of a plan and those that are simply details. As a result, plans developed by nonhierarchical planners get bogged down in unimportant details. In any plan there are levels of detail that are too picky or too vague and a level of detail that is appropriate for the problem; for example, a too-detailed plan for dinner starts with *Go to the table, sit down, unfold the napkin, pour a glass of water, find matches, light the candles...* A too-vague plan is *Sit down somewhere, have food.* Planning with too many details is a waste of effort, but plans that are too vague do not specify which problem-solving operators should be used; a balance between these extremes is necessary for efficient planning.

To this end, the method of *hierarchical planning* has been implemented in a number of planning systems. The method is first to sketch a plan that is complete but too vague and then to *refine* the vague parts of the plan into more detailed subplans until finally the plan has been refined to a complete sequence of detailed problem-solving operators. The advantage of this approach is that the plan is first developed at a level at which the details are not computationally overwhelming.

Hierarchical planning also takes several forms in these systems. One approach, typified by the ABSTRIPS program (Article II.D6, in Vol. I), is to determine which subgoals are critical to the success of the plan and to ignore, at least initially, all others. (In ABSTRIPS, a *detail* is a subgoal for which a subplan can be found if plans have been found to accomplish goals that are not details.) For example, the problem of buying a piano cannot be solved unless two subgoals are accomplished, namely, *Locate piano* and *Get money.* Thus, an initial plan for buying a piano might simply be *Locate piano, get money, buy piano.* Subsequently, this plan can be *refined* with inessential details, such as *Drive to the store* and *Select piano.* ABSTRIPS plans in a *hierarchy-of abstraction spaces*, the highest of which contains a plan devoid of all unimportant details and the lowest of which contains a complete and detailed sequence of problem-solving operators. The advantage of considering the critical subgoals before the details is that it reduces search: By ignoring details, one effectively reduces the number of subgoals to be accomplished in any given abstraction space.

Hierarchical planning was implemented in its earliest form by Newell and Simon (1972, pp. 429–435) in their GPS model of theorem proving in logic. The GPS approach was slightly different from that of ABSTRIPS. In ABSTRIPS, a hierarchy of abstraction spaces is defined by treating some goals as more important than others, while in GPS there was a single abstraction space defined by treating one representation of the problem as more general than others. GPS planned in an abstraction space defined by replacing all logical connectives by a single abstract symbol. The original *problem space* defined four logical connectives, but many problem-solving operators were applicable to any connective. Thus, it could be treated as a detail and abstracted out of the formulation of the problem. A problem could be solved in the abstraction space, the space with only one connective, and the solution could be mapped back into the original four-connective space.

Subsequent implementations of the hierarchical planning approach such as NOAH (Article XV.D1) and MOLGEN (Article XV.D2) are, again, slightly different from either ABSTRIPS or GPS. ABSTRIPS abstracted critical goals, and GPS abstracted a more general representation of an aspect of its problem space. NOAH abstracts problem-solving operators; it plans initially with generalized operators that it later refines to problem-solving operators given in its problem space. MOLGEN goes one step further, abstracting both the operators and the objects in its problem space. In all cases, however, hierarchical planning involves defining and planning in one or more abstraction spaces. A plan is first generated in the highest, most abstract space. This constitutes the skeleton onto which details are fleshed out in lower abstraction spaces. Hierarchical planning provides a means of ignoring the details that obscure or complicate a solution to a problem.

A third approach to planning also makes use of skeleton plans but, unlike hierarchical planning, these skeletons are recalled from a store of plans instead of generated. This approach was adopted in one of the MOLGEN systems (Article XV.E). The stored plans contain the outlines for solving many different kinds of problems. They range in detail from extremely specific plans for common problems to very general plans for broad classes of problems. The planning process proceeds in two steps: First a skeleton plan is found that is applicable to the given problem and then the abstract steps in the plan are filled in with problem-solving operators from the particular problem context. This instantiation process involves large amounts of domain-specific knowledge, often working through several levels of generality until a problem-solving operator is found to accomplish each skeleton-plan step. If a suitable instantiation is found for each abstracted step, the plan as a whole will be successful.

This approach has much in common with that of Schank and his colleagues (see Article IV.F6, in Vol. I). Their approach to natural-language understanding is to use stored *scripts* (and other, more sophisticated structures) to provide top-down expectations about the course of a story.

A fourth approach to planning has been found by Hayes-Roth and Hayes-Roth in human planning (see Article XI.C). It is described as *opportunistic* and is characterized by a more flexible control strategy than is found in the other approaches. The Hayes-Roths have adopted a *blackboard* control structure to model human planning. The blackboard is a "clearinghouse" for suggestions about plan steps, suggestions that are made by planning *specialists*. Each specialist is designed to make a particular kind of planning decision. Specialists do not operate in any particular order; the asynchrony of planning decisions that are made only when there is reason to do so gives rise to the term *opportunistic*. In the Hayes-Roths' model, and apparently in human planning, the ordering of operators that characterizes a plan is developed piecewise—the plan "grows out" from concrete clusters of problem-solving operators.

Opportunistic planning includes a *bottom-up* component, since it is driven by opportunities to include detailed problem-solving actions in the developing plan. It contrasts with the *top-down* refinement process characteristic of hierarchical planning, in which detailed problem-solving actions are not decided until the last possible moment in developing the plan. Another difference between opportunistic planning and other forms is that it can develop *islands* of planning actions—parts of a plan—independently, while hierarchical planners try to develop an entire plan at each level of abstraction. (See Chap. V, in Vol. I, for a discussion of island driving in speech understanding.)

The Hayes-Roths' model is discussed in Chapter XI, on models of cognition, since it is intended as a model of human planning abilities.

Search and the Problem of Interacting Subproblems

Two major, interrelated issues will keep reappearing in this chapter. They are the problem of limiting search and the problem of interacting subproblems. The problem of search is to find an ordering of problem-solving actions that will achieve a goal when there are a huge number of orderings possible, most of which will not achieve the goal. This problem has been called *combinatorial explosion,* since the number of combinations of problem-solving operators increases exponentially with the number of operators (see Chap. II, in Vol. I). The problem of interacting subproblems arises whenever a problem has a *conjunctive goal,* that is, more than one condition to be satisfied. The order in which conjunctive goals are to be achieved is sometimes not specified in the problem, but it can be critical to finding a solution. Sometimes interactions of this sort prevent any solution; for example, if a conjunctive goal is to paint a ladder and paint a ceiling, the second goal *must* be achieved before the first, because one cannot stand on a freshly painted ladder to paint a ceiling. Unfortunately, this information is sometimes not given in the problem but must be inferred.

The problem of search is related to the problem of interacting subproblems because additional search results from premature commitment to an arbitrary ordering of interacting subgoals. In the ladder example, a planner that arbitrarily decided to paint the ladder first would need to *backtrack* and change its plan when it discovered it could not paint the ceiling. Backtracking involves replanning from the *choice point* that failed, in this case, the choice between painting the ceiling and painting the ladder. Backtracking can be very costly.

Interactions between subgoals have been called *constraints* (Stefik, 1980; see also Article XV.D2). They can be inferred from the *preconditions* of operators if the preconditions are explicit. For example, if the operator *Paint ceiling* has several preconditions such as *Have paint, Have brush,* and *Have ladder*, an intelligent planner will infer from these that painting the ladder cannot precede painting the ceiling. A less intelligent planner may construct a plan to paint the ladder first and then realize that it cannot continue; it may then attempt to reorder its actions.

Some of the earliest planners generated initial plans that violated ordering constraints and then tried to go back and fix the plan. They include HACKER, INTERPLAN, and Waldinger's system, all discussed in Article XV.C. These systems applied a powerful heuristic called the *linear assumption*, namely, that

> subgoals are independent and thus can be sequentially achieved in an arbitrary order. (Sussman, 1973, p. 59)

In a historical perspective, this can be seen to be an important heuristic. The number of orderings of problem-solving operators is the factorial of the number of operators, so it is obvious that a problem solver cannot successfully examine *all* orderings in the hope of finding one that does not fail because of interacting operators. The linear assumption says that in the absence of any knowledge about orderings of operators, assume that one ordering is as good as any other and then fix any interactions that emerge. The three programs mentioned above all fix plans by reordering the component operators.

The linear assumption is used in cases where there is no a priori reason to order one operator ahead of another. An alternative assumption is that it is better *not* to order operators than to order them arbitrarily. This assumption arises in slightly different forms in the NOAH planning system (Article XV.D1) and one of the MOLGEN systems (Article XV.D2). NOAH establishes *partial orders* of problem-solving operators by considering their preconditions. For example, it may know that the goal of buying coffee beans has the subgoals *Go to coffee store* and *Get money*, but initially it does not commit itself to an ordering of these operators. However, when it expands each of these goals, it notices that a precondition of getting money, *Be at bank*, interferes with the goal of being at the coffee store; thus, it decides to get money before it goes to the coffee store. NOAH orders operators only to eliminate problems that might arise from picking an arbitrary ordering. MOLGEN also will not order operators until constraints are available to guide it; furthermore, MOLGEN avoids committing itself to using operators or objects without constraints because premature commitment may conflict with other parts of its plan.

The *least-commitment* approach of NOAH and MOLGEN contrasts with the linear assumption, which says, *Commit yourself to any order of operators and then fix it.* This approach works because NOAH and MOLGEN are able to infer constraints that hold between operators. An important aspect of the approach is that it is *constructive;* since planning decisions are made only when the planner is sure they will not interfere with past or future decisions, the planner need never *backtrack* and undo a bad decision. In fact, both of these planners do make bad decisions and can backtrack, but the major research effort has been to avoid backtracking.

Interestingly, human planners do not always use the least-commitment strategy and, consequently, they must sometimes backtrack. Humans *opportunistically* plan to execute an operator when it is convenient to do so. For example, a human may plan to pick up groceries on the way to a football game because it is convenient to do so. Later he (or she) will realize that the groceries will wilt during the game and he will have to replan to avoid this.

Conclusion

We have discussed the structure of plans, concentrating especially on the hierarchical relation between goals and subgoals. When problem solving is discussed in terms of search, it becomes evident that, although finished plans are usually linear or partial orders of problem-solving operators, the search spaces in which the plans are developed are hierarchical. This is because problem-solving operators have preconditions that are subproblems with preconditions of their own, and so on. The term *hierarchical* was shown to refer to two related concepts: Most plans have a *hierarchical structure*, but only hierarchical planners use a *hierarchy of abstraction spaces* to develop a plan.

We have introduced four approaches to planning: nonhierarchical planning as practiced by STRIPS and HACKER; hierarchical planning of the sort done by ABSTRIPS, NOAH, and MOLGEN; script-based planning; and opportunistic planning. Most will be discussed in subsequent articles, although opportunistic planning is covered in Chapter XI, on models of cognition. Nonhierarchical planners are discussed in Article XV.C after a comparison of hierarchical and nonhierarchical planning illustrated by ABSTRIPS and

STRIPS in Article XV.B: NOAH is discussed in Article XV.D1; and the last two articles are devoted to the MOLGEN systems (Articles XV.D2 and XV.E).

The major issue for any planning system is reducing search; instrumental in this are methods for minimizing the effects of interacting subproblems. In particular, the least-commitment approach that derives from hierarchical planning is *constructive,* that is, it requires little or no backtracking.

References

Sacerdoti (1979) is an interesting overview and attempt to taxonomize planning methods. Stefik's (1980) doctoral thesis discusses and compares many planning systems and methods. The references mentioned in this article are representative of the planning literature and provide a readable historical background; one important reference that was not mentioned earlier is Bobrow and Raphael's (1974) review of AI programming languages. Planning has received some attention in cognitive science, and human planning has been examined in AI. References include Schank and Abelson's (1977) book on scripts and plans. Feitelson and Stefik's (1977) study of human experiment-planning. Friedland's (1979) doctoral dissertation on script-based planning, and the research of Barbara and Frederick Hayes-Roth on opportunistic planning (Hayes-Roth, 1980).

B. STRIPS AND ABSTRIPS

HIERARCHICAL PLANNING in the context of the STRIPS and ABSTRIPS planners is the subject of this article (see also Fikes and Nilsson, 1971; Fikes, Hart, and Nilsson, 1972; Sacerdoti, 1974; Articles II.D5 and II.D6, in Vol. I). The two systems are virtually identical except that STRIPS plans in a single abstraction space while ABSTRIPS plans in a hierarchy of them. We present here a single problem—getting a cup of coffee—and show how each of the systems would solve it.

Let us first characterize a problem solver as a program that explores the states that arise from the application of problem-solving operators in search of one that qualifies as a solution to the problem. (Other characterizations of search in problem solving are possible; see Articles II.B1 and II.B2, in Vol. I, for a discussion of *state-space* and *problem-reduction search*.) The first state examined by a problem solver is the *starting state*, and if the problem solver is successful, the last state examined will be the *goal state*.

Problem solvers have available a set of problem-solving operators and objects. When problem-solving operators are executed, they bring about changes in the state of the world. Consider now the problem of getting a cup of coffee. You go to the kitchen and if coffee is made, you pour some. If not, you make some or go out to buy some. If you decide to make some, but there are no coffee beans or ground coffee, you go to the store to get some. If you have no money, you go to the bank first. The relevant operators and objects are:

Operator	*Object*
Boil water	boiling water
Pour X	kitchen
Buy X	coffee-bean store
Make coffee	coffee beans
Go to X	brewed-coffee store
Get money	bank
	money

Each operator has *preconditions* that must be true before that operator can be executed—for example, if there is no coffee to pour, you must make some. Making a precondition true is a subproblem. Because problem-solving operators usually have preconditions, a developing plan usually has a hierarchical structure.

The operators for this problem can be represented in such a way that their preconditions and effects are explicit:

Operator	*Precondition*	*Effect*
Pour coffee	Have brewed coffee	Problem solved
Make coffee	Have beans Have grinder Have boiling water Be in the kitchen	Have brewed coffee
Buy something	Be at store Have money	Have something
Go someplace	Place exists	Be at place Not at any other place
Get money	Be at bank	Have money
Boil water	Be in the kitchen	Have boiling water

The starting state and goal state of the problem can be expressed in these terms also:

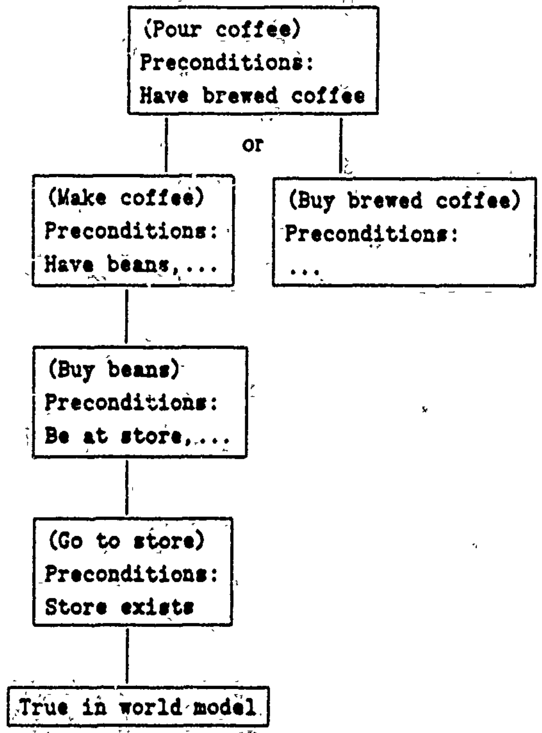
Starting state	*Goal state*
Not have brewed coffee	Have brewed coffee
In kitchen	In kitchen
Have grinder	Have grinder
Have money	Have money
Have boiling water	Have boiling water

If a problem solver knows how each problem-solving operator changes the state of the world and knows the preconditions for an operator to be executed, it can apply a technique called *means-ends analysis* to solve problems (see Article II.D2, in Vol. I, and Article XI.B). Briefly, this technique involves looking for a *difference* between the current state of the world and a desired state and trying to find a problem-solving operator that will reduce the difference. This continues recursively until the desired state of the world has been achieved. STRIPS and ABSTRIPS, and most other planners, use means-ends analysis.

The next few paragraphs illustrate how STRIPS might solve the problem of getting a cup of coffee. First, it compares the starting state and the goal state and immediately finds a difference: *Have brewed coffee.* So it looks for an operator that has *Have brewed coffee* in its list of effects. It finds two: *Make coffee* and *Buy something*, where *something* is instantiated with *brewed coffee.* STRIPS must choose one of them; choosing the first makes the example more interesting, so we will assume it does that.

To make coffee, the four preconditions of the *Make coffee* operator must be fulfilled. STRIPS compares the current state of the world (the starting state) with the first precondition and immediately finds a difference, *Have beans.* Consequently, it goes back and tries to eliminate this difference by searching for an operator that has as its effect *Has beans.* Only one operator applies, namely. *Buy something*, where *something* is instantiated with *beans*. Once again. STRIPS compares the preconditions of the proposed operator with the current state of the world. It notes that the condition *Be at store* is not satisfied. so it must repeat the search once again and find an operator that will get it to the st re. There is such an operator, *Go to someplace*, with the single precondition that the place exist; since the store is one of the objects available to STRIPS, the operator can be executed.

At this point, a plan for solving the problem would have the following hierarchical structure:



Note that executing the operator *Go to store* changes one aspect of the state of the world. The starting state is *In the kitchen*, but *Go to store* changes this to *At the store*. This change satisfies one of the preconditions of the *Buy beans* operator; STRIPS checks the other precondition, *Have money*. Since this precondition is true in the current state of the world, STRIPS is free to execute the *Buy beans* operator. Its execution fulfills the first precondition of the *Make coffee* operator. Furthermore, STRIPS finds the next two preconditions, *Have grinder* and *Have boiling water*, true in the current state of the world. However, the last precondition, *Be in kitchen*, has been made false by going to the store, so before making coffee, STRIPS must find an operator with the effect of making *Be in kitchen* true again. This is simply *Go to kitchen*, and since it has no preconditions it is immediately applicable. Its execution fulfills all the preconditions of *Make coffee;* consequently, that operator can be executed, fulfilling the single precondition of *Pour coffee* and solving the problem.

The final plan for getting coffee is, thus, *Go to the store, buy beans, go to the kitchen, make coffee, pour coffee.*

Means-ends analysis is a powerful problem-solving method because it reduces the amount of search done by a problem solver. At any point prior to solving a problem, one or more goals must be satisfied. Means-ends analysis recognizes only one type of goal, namely, to reduce a difference between two states. Moreover, an assumption of the method is that problem-solving operators can be classified according to the kinds of differences they reduce. Consequently, only a fraction of the available operators will be applicable to any given goal, and search among the operators for an applicable one will be reduced.

Search and Backtracking

One difficulty with means-ends analysis is that it can still develop large search spaces. Although it restricts the number of operators that apply to a goal, there may still be several applicable operators and no a priori basis for selecting one. Moreover, there is no way of knowing whether the subgoals of an operator can be satisfied or whether their evaluation may eventually lead to a dead end, that is, to a subgoal that cannot be satisfied. For example, if the *Go to someplace* operator had a precondition *Have car* but no car existed, all of the processing that led to that operator would have been in vain and the problem solver would have had to *backtrack* to find an alternate path. In the example above, the only other path involves trying to *Buy brewed-coffee,* and it, too, will fail for the same reason. In more complicated problems, one might expect to find several alternative paths that might accomplish a given subgoal, and a substantial amount of backtracking may be needed to solve the problem. Backtracking can be very expensive, so recent planners have been designed to avoid it as much as possible.

Backtracking arises from premature commitment to a problem-solving path. As an illustration, consider again the problem of getting coffee. Assume for a moment that the objects that are available to STRIPS are *kitchen, bank, coffee-bean store, brewed-coffee store.* The *grinder* and the *grinder store* are missing. To solve the problem, STRIPS builds a *search tree,* as shown in Figure B-1.

Briefly, STRIPS would reason that to pour coffee, it must make some or buy some. It opts to make some. To do so, it needs beans, for which it needs money and a bean store. To get money, it must get to a bank, for which a bank must exist. Since a bank does exist, STRIPS plans to go there and get money. It then explores the possibility of going to a bean store; since such

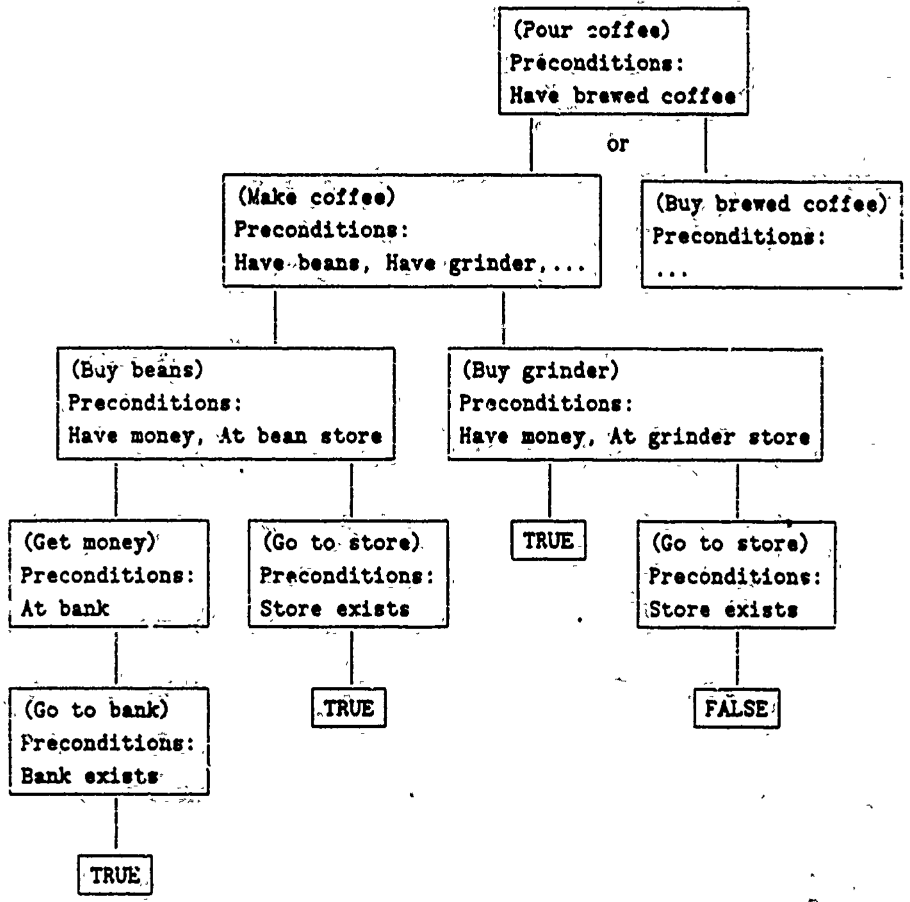


Figure B-1. A search tree for the problem of pouring coffee.

a store exists, STRIPS can go there. Both preconditions for buying beans are fulfilled, so it plans to buy them and then goes on to consider the next precondition of making coffee, which is having a grinder. Since it does not have one, it decides to buy one, for which the preconditions are having money and being at a grinder store. It has money from its previous visit to the bank, so it plans to go to the grinder store. Unfortunately, no such store exists. All of this processing has been in vain—STRIPS cannot possibly make coffee. Its only option is to backtrack to a *choice point* in the plan and try another path. In this case, it can try to buy some brewed coffee. This part of the plan is not illustrated, but it will succeed since a brewed-coffee store exists.

Part of the expense of backtracking in the previous example derives from planning several operations that are actually unimportant details. Intuitively, one would expect STRIPS to have checked much earlier in the plan to see whether a grinder store existed. Similarly, if STRIPS knew that certain stores existed, it should not have worried about how to get to them until later in the plan: getting to places seems like a detail. One would expect a planner first to plan to do all the important steps in a plan and then to fill in the less important ones after it has sketched out the others. In fact, this method is called *hierarchical planning;* the first planner to use it was an extension of STRIPS called ABSTRIPS. We will now briefly describe how it works.

ABSTRIPS plans in a *hierarchy of abstraction spaces.* An ABSTRIPS abstraction space contains all of the objects and operators given in the initial specification of the problem (called the *ground space*), but some preconditions of some operators are judged to be more important than others. For example. *Have boiling water* seems like an unimportant precondition of making coffee because it is so easy to accomplish. Other preconditions such as *Grinder store exists* seem very important, because if they are not true in the ground space, there is no operator that the problem solver can execute to make them true. Preconditions are assigned importance levels, called *criticalities.* When ABSTRIPS starts planning, it plans to achieve only those preconditions that have the maximum criticality—just those preconditions that are critical to the success of the plan. It plans in the *highest* abstraction space. Next, it plans to achieve the preconditions of the steps in its high-level plan that have the next criticality level, and so on, until all the preconditions in a plan have been achieved.

The first step in this process is assigning criticalities. The method used in ABSTRIPS is for a human to draw up a partial ordering of preconditions by intuitively judging their importance; then ABSTRIPS follows an algorithm to adjust the criticalities further. One might guess that the most important precondition is that a place exist, since if it does not, operators that depend on its existence cannot be used in a plan. One might judge having something as the next most important precondition and being somewhere the least important:

Precondition	*Intuitive criticality*
Place exists	3
Have something	2
Be somewhere	1

ABSTRIPS adjusts these criticalities as follows: All preconditions whose truth values cannot be changed by any operator are given a maximum criticality. For each of the other preconditions, if a short plan can be found to achieve it—assuming the previous preconditions are true—it is assumed to be a detail and is given a criticality equal to that specified in the partial ordering. If no such plan can be found, it is given a criticality greater than the highest one in the partial order.

The preconditions *Bank exists, Bean store exists,* and *Brewed coffee store exists* are all assigned a maximum value, say, 5, because their truth cannot be changed by any operator. The four *Have something* preconditions are *Have beans, Have grinder, Have boiling water,* and *Have money;* three of them can be achieved by a short plan, given that the previous preconditions are true. For example, given that the bank exists, a short plan can be found to achieve the precondition *Have money.* These three preconditions are therefore assigned their partial-order rank of 2, and the fourth, *Have grinder,* which cannot be achieved by a simple plan because no grinder store exists, is given the rank of 4, higher than any partial-order rank. Lastly, the *Be somewhere* preconditions are ranked, and since they can all be achieved by simple plans, they are assigned their partial-order rank of 1:

Precondition	*Criticality*
Bean store exists	5
Brewed-coffee store exists	5
Bank exists	5
Have grinder	4
Have beans, boiling water, money	2
Be at brewed-coffee store, bean store, bank	1

ABSTRIPS now formulates a plan in an abstraction space of criticality 5. This means that at this level, any precondition of an operator that has a smaller criticality value is assumed to be true. At this level, ABSTRIPS finds two plans to get coffee: *Make coffee* and *Buy brewed coffee.* It then expands the *Make coffee* plan in an abstraction space of criticality 4, since the *Have grinder* precondition emerges at this level. ABSTRIPS tries to find a subplan for getting a grinder but cannot. Consequently, it recognizes immediately that its level 5 plan to make coffee will fail. It backs up to level 5 again, picks the alternative plan to buy brewed coffee, and pursues it. Figure B–2 shows a trace of its operation in the five abstraction spaces.

In this trace, ABSTRIPS first plans to make coffee, but this plan fails in the abstraction space of level 4. Thus, it backtracks to level 5 and plans to buy brewed coffee. This plan is not expanded further until level 2, when the precondition of having money becomes apparent. At level 1, a precondition of getting money is found, namely, *Be at bank,* and a precondition of buying coffee is found, namely, *Be at store.* ABSTRIPS plans to go to these places; its final plan is *Go to bank, get money, go to coffee store, buy brewed coffee.*

ABSTRIPS solves problems with much less searching and backtracking than STRIPS because it is a hierarchical planner. It generates a hierarchy of plans in which the highest level plans are very sketchy and the lowest level plans are detailed. Since a complete plan is formulated at each level of abstraction before the next level is considered, ABSTRIPS can find dead ends early, as it did with the problem of finding a coffee grinder. The details of the other parts of the plan to make coffee, for example, boiling water and

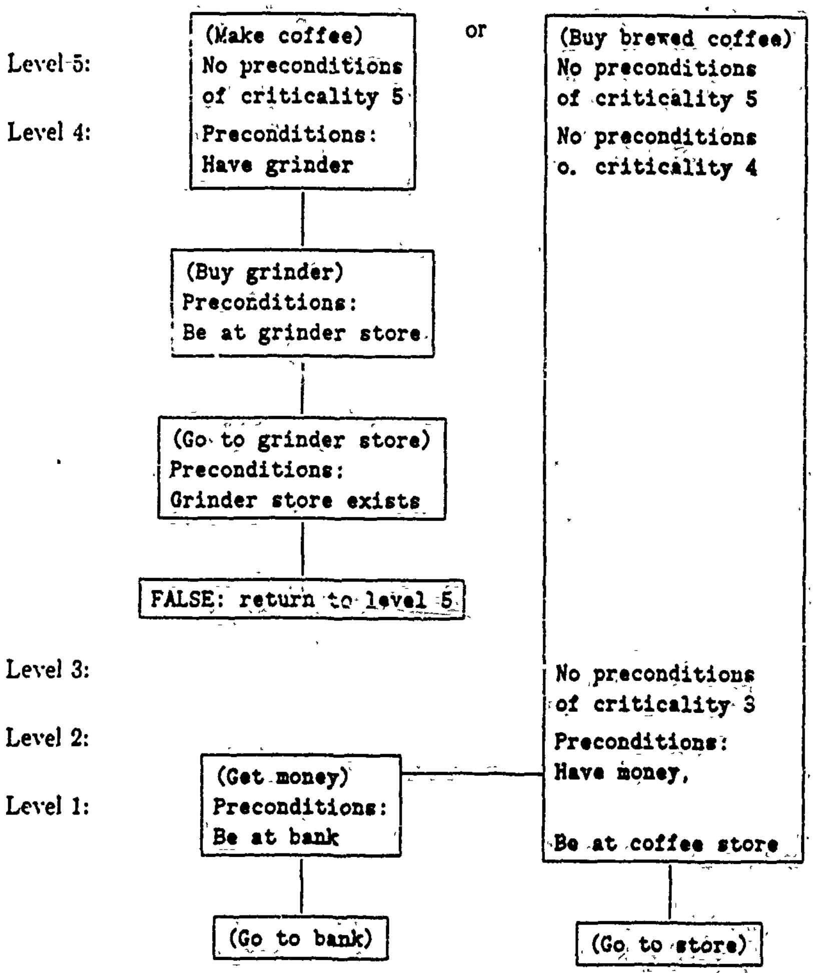

Level 5:
Level 4:
Level 3:
Level 2:
Level 1:
(Make coffee)
No preconditions of criticality 5
Preconditions: Have grinder
or
(Buy brewed coffee)
No preconditions of criticality 5
No preconditions o. criticality 4
(Buy grinder)
Preconditions: Be at grinder store
(Go to grinder store)
Preconditions: Grinder store exists
FALSE: return to level 5
No preconditions of criticality 3
Preconditions: Have money,
Be at coffee store
(Get money)
Preconditions: Be at bank
(Go to bank)
(Go to store)

Figure B-2. A trace of ABSTRIPS in five abstraction spaces.

buying beans, were never considered because ABSTRIPS quickly detected that an important precondition of making coffee could not be satisfied.

References

STRIPS is discussed in Fikes and Nilsson (1971); in Fikes, Hart, and Nilsson (1972); and in Article II.D5 in Volume I of the *Handbook*. ABSTRIPS is discussed in Sacerdoti (1974) and in Article II.D6 (also in Vol. I).

C. NONHIERARCHICAL PLANNING

NONHIERARCHICAL approaches to planning order operations at a single level of abstraction, in contrast to hierarchical planners, which develop entire plans at multiple levels of abstraction. A nonhierarchical planner typically develops a hierarchy of subgoals, but they are all at the same level of abstraction.

The systems discussed in this article are HACKER, INTERPLAN, and the planner developed by Waldinger. They are three attempts to solve the difficult planning task of achieving conjunctive subgoals that are not independent. Many problems are formulated as a conjunction of goals; for example, *spring cleaning* may involve sweeping, washing the floor, washing the windows, beating the rug, and so on. However, these goals are not independent: they cannot be achieved in an arbitrary order. Washing the floor before sweeping is a doomed and grubby operation; a *precondition* of washing the floor is that it be swept clean of loose dirt. Similarly, one should not beat the rug after sweeping, because dragging a dusty rug outside will make the floor dirty and ruin the effect of sweeping. In the terminology of this chapter, beating the rug after sweeping would constitute a *violation of a protected goal,* the goal being a freshly swept house. Similarly, achieving some goals can actually prevent the accomplishment of others, as when washing the floor prevents one from walking across it or using it for any other purpose until it is dry. To any person with minimal housecleaning experience, it will be obvious how and why spring-cleaning tasks must be ordered to avoid their mutual interference, but simple planning programs do not have a priori knowledge about the order in which goals should be accomplished. The problem for these planners is to construct, in the absence of this knowledge, an efficient plan for achieving conjunctive goals that are not independent.

The approach taken by HACKER and INTERPLAN is to formulate plans that are flawed by interferences between subgoals and then to fix them by reordering problem-solving operations in the plan. Waldinger's system is more constructive: Instead of reordering operations in a flawed plan, it develops the plan by inserting operations one by one, checking each for potential interference with established operations.

HACKER and INTERPLAN apply a simplifying heuristic called the *linear assumption* to restrict the number of goal orderings that it considers. It was originally formulated by Sussman (1973) in these terms:

Subgoals are independent and thus can be sequentially achieved in an arbitrary order. (p. 59)

Of course, this assumption is false for many problems, but it does avoid the problem of searching for an ordering of subgoals in which none interferes. The search space of orderings can be enormous, since it grows with the factorial of the number of subgoals in a plan; for example, there are over 3 million distinct orders in which 10 conjunctive subgoals can be achieved. The linear assumption commits the planner to an arbitrary ordering of subgoals rather than searching for an optimal one and, in the event that the ordering is suboptimal, the planner tries to fix it. (For an alternative, *least-commitment*, approach, see the following two articles.)

HACKER

HACKER was developed as a model of skill acquisition by Gerald Sussman at M.I.T. Sussman defines *skill* as a set of procedures, each of which solves a certain kind of problem from the domain of the skill. If a skill does not include a procedure to solve a problem, a new procedure must be designed. Typically, it implements old procedures as a means of achieving subgoals of the new problem. New procedures can turn out to have "bugs" and not work in all the situations for which they are designed, in which case they can be patched to make them work. Often, bugs can be abstracted: that is, within the domain of a skill there are common bugs that show up in many procedures. One very general bug, the one addressed by all the systems in this article, is found in cases in which *conjunctive subgoals* are to be achieved: Achieving one subgoal may prevent the accomplishment of another. Sussman reasons that this bug (and others) is so common that a model of skill acquisition should know how to *debug* the procedures it designs. HACKER is able to do so in many cases.

Although HACKER was designed as a model of skill acquisition, it is interesting in the context of planning because the procedures it develops for solving problems are plans and because the debugging of plans was considered a useful problem-solving technique. For the purposes of this chapter, we will ignore what HACKER contributes to the subject of learning (for this, see Article XIV.D5c) and concentrate on those aspects of skill acquisition that are relevant to planning.

HACKER was written at a time when *procedural* representations of knowledge were popular (see Chap. III, in Vol. I, on knowledge representation). One result of this is that HACKER's various functions are difficult to separate. Rather than explain their extensive interactions, the functions and the knowledge that supports them are described here in general terms. Those of immediate interest are the *answer library*, which contains problem-solving procedures; the *knowledge library*, which contains facts about the domain; the *programming-techniques library*, which is used to propose problem-solving procedures when appropriate ones are not found in the answer library; and several libraries of bugs and appropriate patches.

Problem solving in HACKER would be much like that in PLANNER (see Article VI.A, in Vol. II) were it not for the need to debug plans. PLANNER had only one mechanism for recovering from a flawed plan, namely, *backtracking.* This was very expensive in terms of search time. In contrast, HACKER proposes a plan and then corrects errors in it with programs that are experts in debugging, rather than by backtracking to the point of failure in a plan and blindly trying another problem-solving operation.

The bug that concerns us here is called *prerequisite-clobbers-brother-goal* by Sussman: it arises from the *linearity assumption.* There are often interactions between goals such that achieving the prerequisites for one goal prevents the accomplishment of another. HACKER can solve some of these interaction problems, but sometimes the solution is not optimal. A popular problem for planners is shown in Figure C–1.

HACKER attempts to solve this problem by finding a procedure in its answer library that matches the pattern of the goal: `(MAKE (ON B C))`. It finds a procedure that says,

```
(TO (MAKE (ON X Y))
    (PUTON (X Y))) ;
```

that is, to get block *B* on block *C*, execute the simple procedure PUTON with *B* and *C* as arguments. When it simulates the execution of this program, it discovers that it fails, because *A* is on *B*. A bug in the proposed plan has been found; HACKER now attempts to patch it up. First, a library of types of bugs is consulted, from which HACKER concludes that the bug is a PREREQUISITE-MISSING type. We will not go into the details of this classification. HACKER knows that a prerequisite to one of its planned actions is missing, but it does not know which prerequisite. In its knowledge library it finds several potentially pertinent facts. One is

```
(FACT (PREREQUISITE (PUTON (X Y) (PLACE-FOR X Y)))) .
```

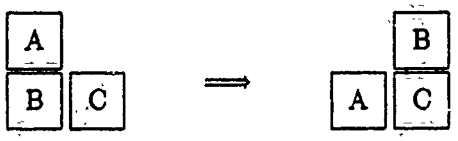


Figure C–1. A planning problem: Get block *B* from under *A* and put it on block *C*.

That is, to put X on Y there must be a place on Y for X to rest. It checks to see whether there is a place on C for B; since there is, this is not the missing prerequisite. The next fact is more enlightening:

```
(FACT (PREREQUISITE (EXPRESSION (CLEARTOP OBJECT))
                    (HAVE () (MOVES EXPRESSION OBJECT)))) .
```

It says that a prerequisite for moving an object is that the object have a clear top. Since A is stacked on B, this prerequisite is not met for B.

HACKER has discovered the identity of the bug that spoiled its initial plan for getting B on C. It now uses this information to modify the plan, applying general methods for fixing bugs that it has encountered before. One such method says that, to patch a PREREQUISITE-MISSING bug, a procedure for attaining the prerequisite should be inserted into the plan before the prerequisite is needed. The prerequisite to be achieved is (CLEARTOP B). HACKER treats this as a subgoal and returns to the beginning of its problem-solving cycle; it looks in the answer library for a procedure that will achieve the prerequisite. We will assume that this procedure exists; if it did not, HACKER would construct it with the help of its programming-techniques library.

To summarize, HACKER solves problems by searching for a procedure known to be appropriate for such problems. If it finds one but the procedure does not achieve the goal as expected, the reasons for the failure are formalized as bugs. Efforts are then initiated to debug the procedure. At any time during problem solving, HACKER may be required to write procedures to achieve certain goals. These are then tested and debugged exactly like procedures found in the answer library.

There are problems for which HACKER cannot generate an optimal plan. One such problem is shown in Figure C-2 and is discussed in the "Anomalous Situations" chapter of Sussman's thesis (1973).

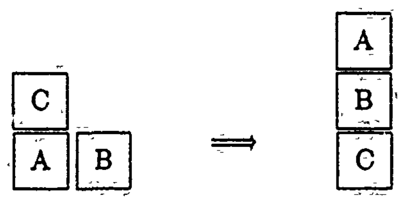


Figure C-2. A problem for which HACKER cannot provide an optimal solution. The proper goal sequence is (CLEAR A), (ON B C), (ON A B).

HACKER knows from previous experience that it is wise to build from the ground up; therefore, for the problem in Figure C–2, it constructs a plan to

```
((ACHIEVE (ON B C))
 (ACHIEVE (ON A B)))
```

But when it simulates execution of this plan, it notices that, after putting *B* on *C*, it must take it off again, and take *C* off *A*, in order to clear *A* for putting *A* on *B*. This constitutes a *protection violation* of the previously achieved goal, namely, (ON B C). HACKER treats protection violations as bugs: unfortunately, this one cannot be fixed simply by reordering its goals. If HACKER tries to solve the problem by achieving (ON A B) and then (ON B C), it finds that, after achieving (ON A B), another protection violation results from trying to (CLEAR B) to put it on *C*. Regardless of the order in which HACKER attempts to achieve the goals of the problem, a protection violation occurs. The only alternative is suboptimal—to permit the violation and then to achieve the violated goal again at a later time, for example, by putting *B* on *C*, then taking it off again, taking *C* off *A*, putting *B* back on *C*, and finally putting *A* on top.

When HACKER discovers a protection violation, it tries to reorder the operations in its plan. However, it is limited to reordering operations at one particular level of the plan; in the previous example it tried to reorder the initial goals. To solve the problem, it is necessary to reorder goals at different levels of the plan. HACKER need not reorder the goals (ON B C) and (ON A B), but it must achieve a *subgoal* of (ON A B), namely, (CLEAR A), before it achieves (ON A B). This kind of reordering of levels of goals is too subtle for HACKER. However, another program called INTERPLAN does consider these more complex reorderings.

INTERPLAN

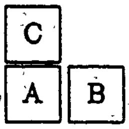
INTERPLAN was developed by Austin Tate at the University of Edinburgh in 1974. It employs a convenient declarative representation called a *tick list* to allow protection violations to be detected easily and to give the system the relevant information for recovery (Tate, 1975a). In the event of a protection violation, INTERPLAN first tries the same reorderings as HACKER; namely, goals are reordered at a single level of the subgoal hierarchy. But if this fails, it considers more general reorderings. In particular, the subgoal at which failure occurred is *promoted*, that is, moved before its superordinate goal, and possibly before other goals as well.

The space of goal orderings considered by INTERPLAN is thus larger than that considered by HACKER, but for this added effort it gains the ability to optimize plans that HACKER could not optimize.

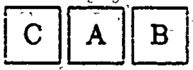
Consider the problem from Figure C–2. INTERPLAN initially proceeds like HACKER:

Goal or action *State*

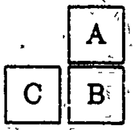
ACHIEVE (ON A B) 1.



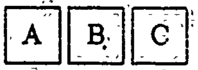
ACHIEVE (CL A)
APPLY (Clear A) 2.



APPLY (Puton A B) 3.

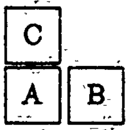


ACHIEVE (ON B C)
ACHIEVE (CL B)
APPLY (Clear B) 4.

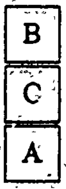


* (1) Protection violation with state 3: Reorder

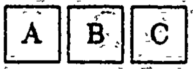
ACHIEVE (ON B C) 1.



APPLY (Puton B C) 5.

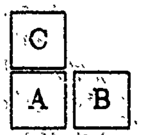
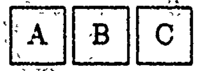
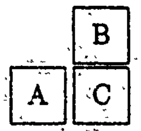
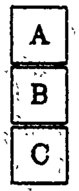


ACHIEVE (ON A B)
ACHIEVE (CL A)
APPLY (Clear A) 6.



* (2) Protection violation with state 5:

At this point in the problem, HACKER resigns itself to a suboptimal plan. It has tried the two possible orderings of the goals (ON A B) and (ON B C), and neither of them produces plans free of protection violations. In order to solve the problem, a *subgoal* of one of the main goals must be achieved before either of the main goals. HACKER is not capable of reordering goals between levels, but INTERPLAN is. It decides to *promote* the subgoal that caused the protection violation; it returns to the starting state of the problem and immediately tries to achieve (CL A):

Goal or action		*State*
PROMOTE (CL A) ACHIEVE (CL A)	1.	
APPLY (Clear A)	7.	
ACHIEVE (ON B C) APPLY (Puton B C)	8.	
ACHIEVE (ON A B) APPLY (Puton A B)	9.	

* (3) Goal achieved

Subgoal promotion is thus a useful method for reordering goals when they interfere with each other. The method and the *tick-list* data-structure that facilitates it are discussed in detail in Tate (1975b).

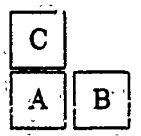
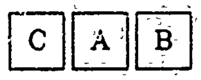
Goal Regression

HACKER and INTERPLAN *backtrack* when they find a protection violation; they reorder a couple of goals and then start planning to achieve them in the new order. For simple problems like the previous example, this method will suffice, but if there are several conjunctive goals, and many or most goal orderings produce subgoal interactions, the method is very inefficient. Moreover, when these planners reorder their goals, all goals affected by the reordering must be achieved again. This can lead to the same solution being achieved for a subgoal a number of times because superordinate goals interacted with each other.

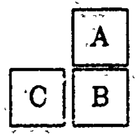
An alternative approach is to construct a plan by solving one conjunctive subgoal at a time, checking that each solution does not interfere with other goals that have already been achieved and moving the offending goal to a different place in the plan if it does. A planner that works this way was developed by Richard Waldinger (1977). He introduced the concept of *goal regression* to handle interference between goals.

At any point in a plan a goal may have been achieved, but after another step it may have been violated. This was illustrated earlier in the problem in Figure C–2: after (ON B C) had been achieved, it was violated to achieve (CLEAR A). Waldinger noted that for any goal *G* and operation *O*, there is no guarantee that *G* will be true after *O*, but that a new goal *G′* can be found such that if *G′* holds before *O*, *G* will hold after *O*. Finding this new goal *G′* is *goal regression*, or *passing the goal back over the operator.* Goal regression can be used to guarantee that goals that have been achieved are not violated by subsequent operations. The basic planning algorithm is to achieve the first of the conjunctive subgoals of the problem and then expand the plan by regressing subsequent subgoals from the end of the plan to a point in the plan where their accomplishment will not violate those previously achieved.

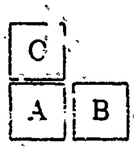
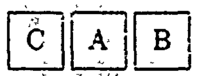
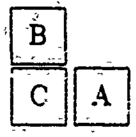
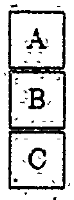
Consider again the three-blocks problem. Waldinger's system can solve the problem regardless of the order in which it approaches the subgoals, but we will illustrate it planning to achieve (ON A B) before (ON B C). First, the system removes block *C* from atop *A* in order to clear *A*. The plan looks like this:

Goal or action		*State*
	1.	
ACHIEVE (ON A B) (Clear A)	2.	

Now the system puts *A* on *B*:

(Put A on B)	3.	

The plan consists of two actions, (Clear A), (Put A on B). The system now attempts its second goal, appending it to the end of the plan. However, it ~~finds that achieving one of its preconditions, (Clear B), violates the protected~~ relation *A is on B*. Rather than reordering the conjunctive goals of the plan, as HACKER and INTERPLAN do, the system simply passes the offending goal back over previously achieved subgoals until it finds a place in the plan where the goal will not interfere with any others. In this case, the goal (ON B C) is moved in front of the action (Put A on B). The plan now looks like this:

Goal or action		*State*
	1.	
ACHIEVE (ON A B) (Clear A)	2.	
ACHIEVE (ON B C) (Put B on C)	3.	
(Put A on B)	4.	

When a proposed operator causes a protection violation, an attempt is made to insert it at earlier points in the plan, checking to see whether the interaction is avoided and to see that no new protection violations occur. However, the choice of where to insert the new operator is not guided by any information. It involves simply searching back in the plan and checking at each position to see if it is suitable. Waldinger's system does not check whether a later step is made redundant by the insertion of the operator, so a less than optimal plan may be produced.

The main advantage of Waldinger's approach is that it is constructive: Plan steps are added one by one, and the only difficulty is finding out where they should go in the plan. This can involve a considerable amount of searching, but it avoids the inefficient repeated achieving of subgoals that HACKER and INTERPLAN must do after reordering.

Conclusion

We have discussed here three nonhierarchical approaches to planning: HACKER, INTERPLAN, and Waldinger's system. Each suffers from interacting subproblems; the first two systems are forced to backtrack and reorder subgoals, and Waldinger's system, though it avoids backtracking by constructive goal regression, must evaluate the consequences of putting a subgoal at a proposed place in a plan. In the remaining articles of this chapter, we will consider hierarchical and script-based planning as alternatives to nonhierarchical planning.

References

HACKER is discussed in Sussman's doctoral thesis (1973; also Sussman, 1975). INTERPLAN is discussed in Tate's thesis (1975b), although his *IJCAI* article (1975a) is more accessible. See Waldinger (1977) for a presentation of his system.

D. HIERARCHICAL PLANNERS

D1. NOAH

IN NOAH, Earl Sacerdoti made some significant advances in problem solving and planning. NOAH (Nets of Action Hierarchies) was designed as part of the Computer-based Consultant project at SRI International, Inc., around 1975 (see Article VII.D2, in Vol. II). It uses a representation for plans called the *procedural net,* which has a richer structure than previous problem solvers. In contrast to these earlier efforts, the procedural net represents both procedural and declarative knowledge about problem solving. The procedural knowledge (also called *domain knowledge*) includes functions that expand statements of goals into subgoals and that simulate the actions of operators that transform one state into another. Declarative, or *plan,* knowledge represents the effects of executing these functions; for example, if a procedure is executed that puts one block on top of another, NOAH records that the supporting block no longer has a clear top surface. Because the effects of actions are represented explicitly, NOAH can reason about them. In fact, NOAH employs a set of procedures called *critics* that are sensitive to those effects of actions that would jeopardize the success of the plan. Critics are used to detect and correct interactions, eliminate redundant operations, and so forth.

Problem solving in NOAH is accomplished by developing the procedural net. From a single node that represents the goal to be achieved, a hierarchy of nodes is developed that represents levels of subgoals to be achieved before the original goal can be accomplished. The original goal node contains a pointer to a set of functions that expand goals into subgoals. When one or more of these functions are executed, subgoal nodes are added to the procedural net. They are linked to the original goal—their *parent*—and to each other, and, like their parent, they contain pointers to functions that expand goals to subgoals. In addition, the nodes representing the subgoals include a declarative representation of the effects, if any, of executing the functions.

After the original goal node has been expanded, there are two levels of representation of the problem, the first of which is the goal node. The second is a series of subgoals that, when achieved, will have the effect of achieving the original goal. These nodes are themselves expanded as their parent was. NOAH continues to add nodes to the procedural net that are more specific versions of the goals represented by their parents. Eventually, the original goal of the problem is replaced by several levels of more detailed goals and, finally, by a level of goals that can be immediately attained by simple problem-solving operators.

Thus, NOAH plans by developing a hierarchy of subgoals. These will sometimes be called *abstract operators*. A distinction is made here, as elsewhere in this chapter, between the simple *problem-solving operators* specified in the problem space and abstract operators that will eventually be expanded to problem-solving operators. Abstract operators are goals, and their expansions are subgoals, in the sense that such operators specify abstract actions that the planner would like to execute but that it cannot execute until they are expanded to subgoals attainable by problem-solving operators.

In addition to abstract and problem-solving operators, NOAH has *planning actions*. These include the functions that expand goals into subgoals and the actions of various critics. They are not part of the emerging plan but, rather, are the actions by which NOAH develops the plan.

Note that whenever NOAH expands a goal to subgoals, it runs the risk of creating *interacting subproblems* (see Article XV.C). This problem arises when a planner commits itself to an arbitrary order for achieving conjunctive goals. NOAH avoids the problem in two ways: first, by not ordering subgoals until there is some reason to do so and, second, by continually examining the developing plan for potential subgoal interactions and correcting them before they arise. This allows NOAH to solve interaction problems constructively: Operators are not ordered until a potential interaction is detected, and then they are ordered to avoid the interaction. This contrasts with the planners in the previous article; those planners ordered operators arbitrarily, and, if an interaction emerged, they backtracked and replanned to try to avoid the interaction. These planners are said to *overconstrain* a plan by committing themselves to orderings arbitrarily; NOAH is said to *underconstrain* a developing plan by not committing itself to any orderings except to avoid an interaction.

Application

NOAH was applied in the domain of assembly tasks, and it proved useful and powerful. It provided instructions to a human apprentice, who then carried out NOAH's plan. The procedural net was well suited to this task, because it allowed a plan to be specified at any of several levels of detail; for example, NOAH could instruct a trained engineer to *bolt the mounting bracket to the frame*—a high-level instruction—but it could tell a novice how to accomplish this goal in detail if necessary. The procedural net also made it easier to monitor the execution of the plan. If an unexpected situation arose, NOAH could replan by patching the procedural net. The building of the plan was kept distinct from its execution, but control could pass from the planner to the execution monitor at any stage.

The Structure of the Procedural Net

The procedural net contains several levels of representation of a plan, each level more detailed than the previous one. Each consists of a *partially ordered* sequence of nodes that represent goals at some level of abstraction. To avoid overconstraining the order in which goals are achieved, NOAH assumes they can be attained in parallel until it has some reason to put one before or after another.

Each node in the procedural net is attached to its more detailed expansion in the next level; for example, the node representing the abstract goal *Make coffee* may be expanded to a handful of more detailed goals, such as *Grind coffee, Boil water, Put the coffee in a filter, Pour the water through it.* NOAH will not commit itself to any particular ordering of these operators until it has reason to do so.

The statement of the problem goal is the top-level node, representing a plan at a very high level. A simple example of the structure of the net with two levels is given in Figure D1–1. The *S* and *J* nodes represent *split* and *join*, respectively; they are dummy nodes that bound actions that are assumed to be executable in parallel. NOAH uses this formalism to represent operations for which it has not chosen an ordering.

NOAH expands a single goal node in the procedural net into a hierarchy of plans at various levels of abstraction. To do this, it uses procedures that expand abstract operators into more detailed ones. Much *domain* knowledge is implicit in these procedures; for example, one such procedure might be:

> *If the abstract operator is* (MAKE COFFEE),
> *then expand it to the operators* (BOIL WATER), (GRIND COFFEE),
> (PUT COFFEE IN FILTER), (POUR WATER THROUGH).

The problem that NOAH is to solve determines what knowledge will be represented in these procedures; the preceding procedure may be appropriate

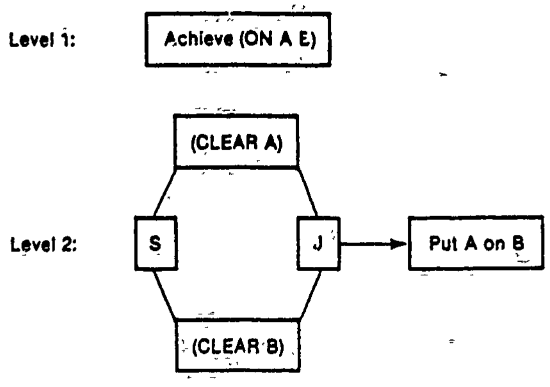


Figure D1–1. An action hierarchy (in a blocks world).

for the *coffee* domain but not for any other. Since these procedures contain so much knowledge about the problem domain, they are called SOUP functions, for Semantics of User Problem. They are written in an extension of QLISP.

Expanding the Procedural Net with SOUP Functions

Consider again the simple blocks-world action hierarchy in Figure D1-1. To achieve it, and to solve simple blocks problems, two SOUP functions are required. One, shown in Figure D1-2, expands any goal of the form (ACHIEVE (ON X Y)), and the other expands any goal of the form (CLEAR X) (these are the only functions required). The main goal of the problem is associated with both functions, since at the outset of the problem it is not known which will apply. However, only (PUTON X Y) matches the *pattern* of the main goal, so only it is applied. (See Article VI.A, in Vol. II, for a discussion of *pattern-directed invocation* of procedures in PLANNER.)

Applying (PUTON X Y) to the main goal of the problem generates three subgoals. The PGOAL forms the basis for constructing subgoals; when a PGOAL is activated, a new node is generated at the next level in the net whose name is the PGOAL's first argument, for example, (CLEAR X). The three PGOALs in PUTON create the nodes (CLEAR A), (CLEAR B), and (Put A on B). The first two are conjunctive, as is specified by the "AND" in the function. NOAH does not choose an order to attain them but assumes they may be attained in parallel and thus surrounds them with *split* and *join* nodes.

The function (PUTON X Y) also specifies the effects of achieving these subgoals. The effects of applying CLEAR to X or Y is to assert CLEARTOP for that

```
(PUTON
  (QLAMBDA (ON ←X ←Y)
  (PAND
    (PGOAL (Clear X)
           (CLEARTOP X)
           APPLY
           (CLEAR))
    (PGOAL (Clear Y)
           (CLEARTOP Y)
           APPLY
           (CLEAR)))
   (PGOAL (Put X on top of Y)
          (ON X Y)
          APPLY NIL)
   (PDENY (CLEARTOP Y))))
```

Figure D1-2. SOUP code for the blocks problem.

block, and the effect of putting X on Y is to DENY the assertion of (CLEARTOP Y). These effects are represented declaratively in the *add list* and *delete list* of a node. The add list is a list of propositions that become true after the goal is achieved, and the delete list represents the propositions that are no longer true after the goal is achieved.

Finally, the SOUP function specifies which other SOUP functions should be applied to expand the subgoals it has just created. It suggests that the appropriate function for the subgoal of clearing A or B is CLEAR. It makes no such suggestion for the third subgoal, Put A on B, because this goal can be accomplished by a single problem-solving operator and need not be further expanded. This mechanism increases the efficiency of problem solving and helps to avoid backtracking. Several SOUP functions might apply to a node in the procedural net, but the parent of the node can specify, at the time the node is created, which function is to be used to expand it. This reduces search. (However, the user may explicitly cause NOAH to consider alternatives by using a POR function inside a SOUP procedure. In this case, alternative expansions are generated in parallel until one is seen to be simpler than the other.)

The Concept of "State" in NOAH

Problem solvers are typically regarded as searching through a space of states for one that qualifies as a solution. One conception of a state in problem solvers like STRIPS and GPS is that a state is a collection of propositions. New states are generated from old ones by the application of operators; that is, operators make some old propositions false and add new true propositions. Eventually, and depending on the power of the problem solver, a state will be generated that includes just the propositions required for the problem to be solved.

NOAH can also be characterized in this way, but the knowledge that makes up a state in NOAH is quite distributed. Some knowledge—that which will never have its truth value changed—is represented in a *world model.* This includes the state of the world that holds when problem solving starts. When some aspect of that state is changed, the proposition describing it is removed from the world model. The changed state of the world is represented by the propositions added to the add list or delete list of the operator that changed the state. Thus, NOAH knows which aspects of its world have not changed—they are represented in the world model—and it distributes its records of changes throughout the procedural net.

Changes are summarized at each level in the net by a *table of multiple effects* (TOME), which contains an entry for every proposition that was asserted or denied by more than one node at that level in the net. TOMEs are used to check for interactions between goals; if a single proposition has its value changed by more than one action in a plan, there is a possibility of interference between the actions.

NOAH uses programs called *critics* to check for interferences. A critic simply checks a TOME for the kinds of conflicts it is designed to correct. When a conflict is found, the critic has a limited number (usually only one) of corrective actions it can take. If all of the critics can successfully eliminate any conflicts found, the next level is expanded. There is presently only a limited ability to backtrack on failure. Three critics are described here.

The RESOLVE-CONFLICTS critic. This examines conjunctive goals that are to be achieved in parallel. If an action taken to achieve one goal removes a precondition of an action in the other, the critic attempts to order the actions so that neither violates a precondition of the other. This critic is similar to the debugging procedure in HACKER for reordering conflicting goals. The important difference is that HACKER backtracks and *reorders* arbitrarily ordered operations, while this critic constructively orders goals that were previously unordered.

The ELIMINATE-REDUNDANT-PRECONDITIONS critic. Sometimes during planning, the same operation gets specified twice when it need be done only once. This critic fixes the problem.

The USE-EXISTING-OBJECTS critic. Formal objects, essentially placeholders, are used whenever there is not a clear choice of what value to give a variable. This critic will substitute a value when a clear choice becomes possible at a lower level of planning.

There are other critics in the system; some have a general purpose like those above, while others are specifically designed for a given domain. More can be added at any time. The critics described here are sufficient for the following example.

Planning in NOAH

The planning algorithm of NOAH operates repeatedly on the current lowest level of the procedural net. Initially, a node is constructed for the goal NOAH is given as its task. All SOUP procedures are available to expand this node; expanded nodes are associated with a much smaller set of SOUP procedures by the procedure that generated them. Once all the nodes in the current level have been expanded to produce a new level, critics check for interactions before another level of expansion is tried.

An Example

This example shows NOAH solving the three-blocks problem that was so difficult for the planners in the previous article.

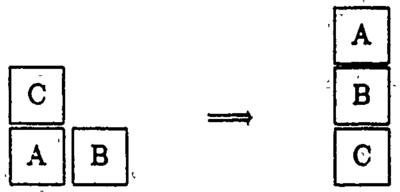
NOAH's *world model* contains the propositions:

```
(ON C A)
(CLEARTOP B)
(CLEARTOP C)
```

This constitutes the *starting state* of the problem. The goal is also written as a proposition:

`(AND (ON A B) (ON B C)).`

Graphically, the starting state and the goal look like this:

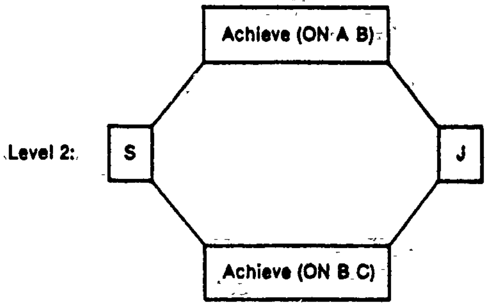


The PUTON and CLEAR functions discussed earlier are used in this problem.

The first node in the procedural net is:

`Level 1:` `Achieve(AND(ON A B)(ON B C))`

This is expanded to two parallel actions by merit of NOAH's policy about conjunctive goals: They are not ordered until there is some reason to do so.



This is a simple expansion; the critics can find nothing to criticize about it. The PUTON function is now used to expand each of the nodes at level 2. (Refer back to Figs. D1-1 and D1-2 for an explanation of how this works.) The result is shown in Figure D1-3.

The RESOLVE-CONFLICTS critic notices that node 3 will delete a precondition of node 6, namely, that *B* is clear (node 4), because node 3 adds a statement to its delete list that DENYs (CLEARTOP B). When a table of multiple effects is compiled for this level, NOAH notices that (CLEARTOP B) is implicated in the effects of both nodes 4 and 6. Since NOAH has not committed itself to achieving any of its goals in a particular order, it need not backtrack to modify its plan in any destructive way. Instead, it uses this conflict as an opportunity to introduce constructively a partial ordering of goals: It decides

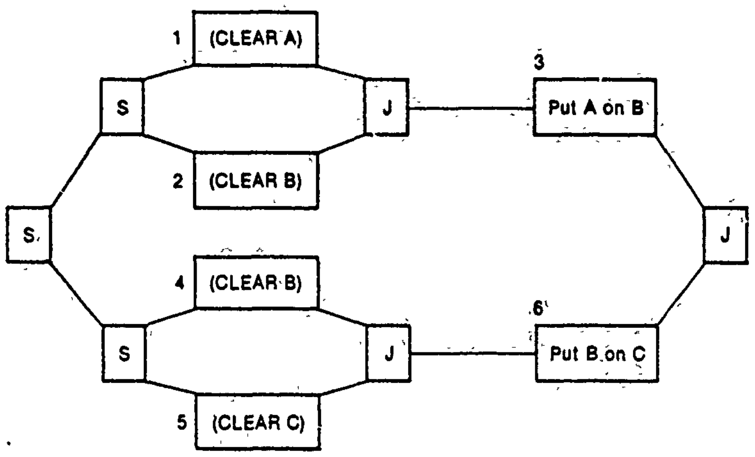


Figure D1-3. Level 3 before criticism, with nodes numbered for reference.

to accomplish node 3 after it has done everything else. Figure D1-4 shows this reordering.

Next, the REDUNDANT-PRECONDITIONS critic observes that nodes 2 and 4 are redundant and eliminates node 2. This step is shown in Figure D1-5.

NOAH next expands the (CLEAR A) goal at level 3. Actually, that is the only goal that remains to be expanded, since *B* and *C* have been clear from the start of the problem, and the (Put X on Y) goals are achieved by simple problem-solving operators. To achieve (CLEAR A), NOAH needs to move *C* off of it and put *C* someplace; it does not know where, so it makes up a placeholder. Block *C* cannot be moved unless it is clear, so the final sequence

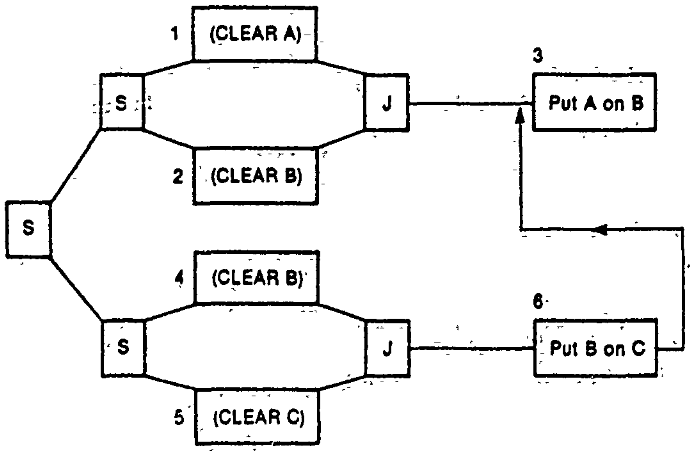


Figure D1-4. Level 3 after the RESOLVE-CONFLICTS criticism.

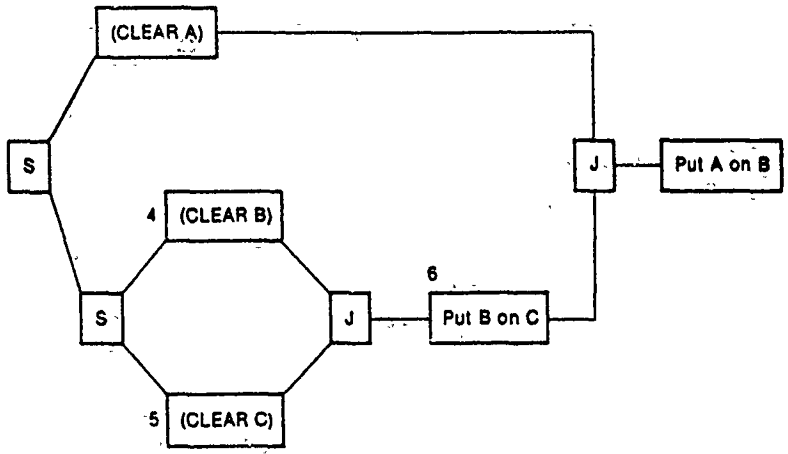


Figure D1-5. Level 3 after all criticism.

that NOAH plans in order to clear *A* is (CLEAR C), (Put C on Object1). This is illustrated in Figure D1-6.

NOAH notices that node 6 may interfere with its latest goal, so the RESOLVE-CONFLICTS critic decides to order node 6 after it has achieved (Put C on Object1). See Figure D1-7.

Finally, the ELIMINATE-REDUNDANT-PRECONDITIONS critic notices that (CLEAR C) is mentioned twice in the plan. It eliminates one of the nodes. The final plan is shown in Figure D1-8.

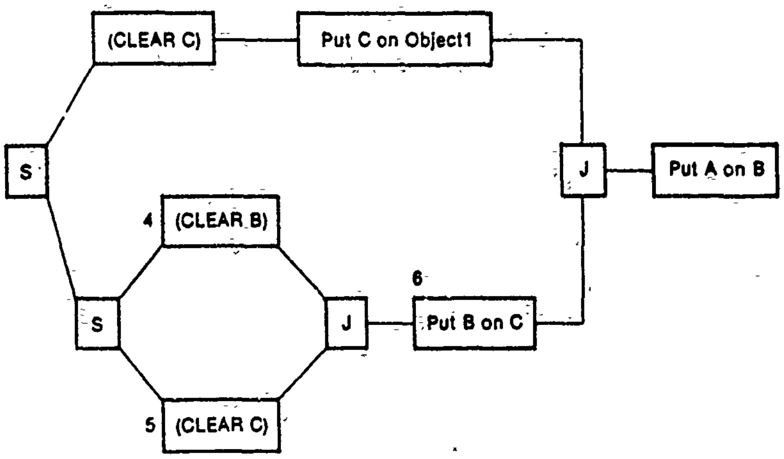


Figure D1-6. Level 4 before criticism.

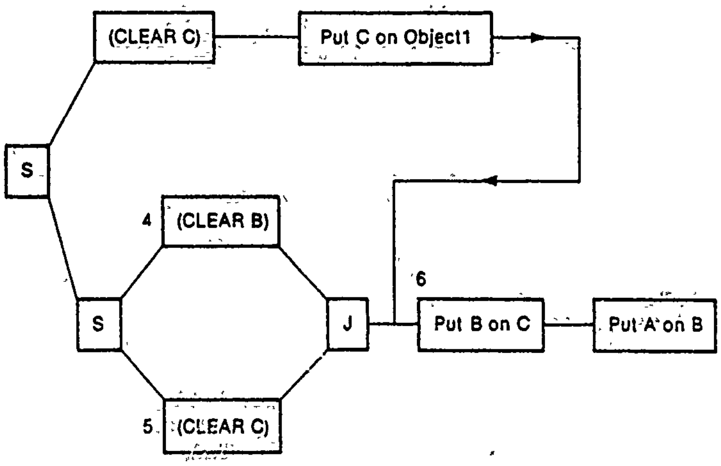


Figure D1-7. Level 4 after the RESOLVE-CONFLICTS criticism.

Conclusion

NOAH plans with a combination of procedural and declarative knowledge. Initially, all NOAH's knowledge is in procedural form—local domain knowledge in the SOUP code and global knowledge in the critics. At the outset of planning, NOAH is given a world model and a goal that it develops into a hierarchical procedural net. As it plans, it records in a declarative form—in add lists and delete lists—knowledge to help it avoid interaction problems. To reason about interactions and possible orderings of goals, this information is summarized in a table of multiple effects. Critics consult these tables after each level has been expanded; they order and alter the plan constructively.

References

NOAH is discussed in detail in Sacerdoti's doctoral dissertation (printed as an SRI technical note, 1975). NOAH has been extended by Tate (1976), and a distributed implementation is discussed by Corkill (1979).

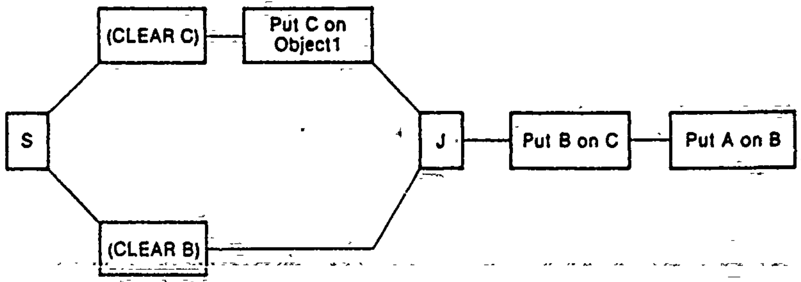


Figure D1-8. Level 4, final plan.

D2. MOLGEN

THE PREVIOUS articles have demonstrated the utility of *problem-reduction* in planning—dividing a problem into subproblems that are more easily solved. But problem reduction has an associated liability, namely, that subproblems are rarely independent. Solving one may prevent solving another. A number of approaches to this problem have been presented in the previous articles. HACKER and INTERPLAN used destructive reordering of subgoals: Waldinger's system employed a more constructive goal-regression method (see Article XV.C). In NOAH (Article XV.D1), the conceptual leap was to avoid linear orderings of subproblems as long as possible and to plan initially with abstract goals that were refined in such a way as to avoid subproblem interactions.

In this article, we discuss the MOLGEN system—a knowledge-based program that assists molecular geneticists in planning experiments. There are actually two MOLGEN planners. one developed by Friedland (1979; see also Article XV.E) and another, the one this article is about. by Stefik (1980). MOLGEN extends the work on hierarchical planning to include a *layered* control structure for *meta-planning*. Plans are constructed in one layer, decisions about the design of the plan are made in a higher layer, and strategies that dictate the design decisions are made at a still higher level. A key idea in MOLGEN is to represent the interactions between subproblems explicitly and declaratively, so that MOLGEN can reason about them and use them to guide its planning. The structure that represents an interaction is called a *constraint*.

Levels of Planning

Control of planning in MOLGEN switches between three *layers*, or *spaces*. The lowest layer, called the *planning space*, contains a hierarchy of operations and objects typical in a gene-splicing experiment. At the lowest level of this layer are bacteria, drugs, and laboratory operations, which are represented by knowledge structures called *units* (Stefik, 1979); generalizations of these include the general objects *gene, organism,* and *plasmid* and the general laboratory operations *merging, amplifying, reacting,* and *sorting.* Initially, MOLGEN plans experiments with these abstract objects and operators. As it chooses specific operators or objects to replace the abstract ones, it introduces constraints into its plan. For example, it plans at an abstract level to *sort* two kinds of bacteria. At a later time, *sort* is replaced by *screen*, which sorts bacteria by killing one group of them with an antibiotic. This decision results in the constraint that the antibiotic be potent against one kind of bacterium but not the other.

The utility of hierarchical planning is illustrated by the preceding example. It shows that although a planning decision to use a particular operation affects later decisions about the kinds of objects to use, this interaction is absent as long as the plan is formulated at an abstract level. Using hierarchical planning, a complete, abstract plan is constructed without attention to these interactions. Then, as steps in the plan are *refined,* the interactions that arise are explicitly represented as constraints and are resolved. The act of refining plan steps involves replacing an abstract operator with a more specific one or replacing an abstract object with a more specific one. If hierarchical planning were not used, every planning decision would introduce interactions; each decision would affect the decisions following it. Early planners like those discussed in Article XV.C produced initial plans that were crippled by interactions and then attempted to reorder planning steps to alleviate them. These planners were said to *overconstrain* their plans; in contrast, MOLGEN and NOAH (see Article XV.D1) produce *underconstrained* plans and add constraints *constructively.*

The middle layer at which MOLGEN plans is called the *design space.* At this level, MOLGEN makes decisions about how its plan is to develop. The operators of the design space dictate steps taken in the design of a plan, for example, proposing a goal or refining an operator. The objects in this space include *goals* and *constraints.* MOLGEN reasons about plans with the objects and operators in the design space, just as it reasons about molecular genetics with the objects and operators in the planning space.

The top layer of planning for MOLGEN, the *strategy space,* includes four very general operators that dictate planning strategy. These are FOCUS and RESUME, which together propose new planning steps and reactivate old ones that have been "put on hold," and GUESS and UNDO, which make planning decisions heuristically when there is not sufficient information to focus or to resume. UNDO is a backtracking operator that undoes decisions that have overconstrained a plan. Much of the research effort in MOLGEN has gone into avoiding backtracking by developing underconstrained plans, but in the rare cases where a guess must be made about a plan step (e.g., choosing the identity of a bacterium), the unforeseen constraints introduced by the choice may force backtracking and a different choice.

Of the three layers of planning in MOLGEN, only the planning space is unique to a domain, in this case, molecular genetics. The design and strategy spaces contain objects and operators that apply to planning in any domain.

Control of Planning in MOLGEN

The three layers discussed above constitute a hierarchically organized control structure for MOLGEN. At the highest level, the strategy space, decisions are made about the style of planning. Two styles are available, *least commitment* and *heuristic.* During the least-commitment cycle, MOLGEN sends a message to the design operators in the design space asking whether they can suggest any tasks to do. Tasks include proposing a goal (after noticing a difference between the current state and the goal state), refining an object or an operator, and formulating a constraint. MOLGEN may fail to find a task for which it has the constraints to proceed successfully; for example, it may propose to refine an object—a bacterium—to a particular species of bacterium, but it may lack the guarantee that this refinement will not interfere with later steps in the plan. In this case, it will *suspend* this step and look for another. If MOLGEN cannot find any design steps to execute immediately, it checks whether any previously suspended steps can be executed; information may have become available since their suspension that justifies their reactivation. The least-commitment cycle oscillates between finding a planning step to execute and reactivating suspended steps in the light of new information. It is called *least commitment* because it will not commit itself to a plan step that might have to be abandoned at some later point in the development of the plan. If MOLGEN cannot find a plan step that satisfies the requirements of the least-commitment cycle, it switches to the heuristic cycle in which it guesses a plan step.

MOLGEN uses three kinds of operations on constraints. The first, called *constraint formulation*, involves identifying interactions between solutions for goals. Often the goals are to refine abstract objects or operators; for example, the goal of sorting two kinds of bacteria is achieved by *screening* one of them with an antibiotic. When this solution is proposed, a constraint is formulated, saying that the choice of bacterium and antibiotic is now constrained by the requirement that one kind of bacterium should be susceptible to the antibiotic.

The second operation with constraints is called *constraint propagation*. This is the creation of new constraints from old ones, which helps refine abstract parts of a plan. For example, the single constraint described above reduces the number of bacteria or antibiotics that MOLGEN is considering, because not all bacteria are susceptible to all antibiotics. Constraint propagation collects other constraints on the bacterium and antibiotic, formulated perhaps in other parts of the plan. As a result of constraint propagation, abstract plan steps that might have been refined in dozens of ways are constrained to have a relatively small number of potential refinements. Often, individual subproblems are constrained to some extent, but not enough to narrow down the space of solutions significantly. However, when the individual constraints on individual subproblems are propagated, the sum of the constraints often eliminates one or more solutions. For example, during a day, a person may have two goals: to get some exercise and to get to school in a short time for a class. The first problem, to get exercise, is constrained only by the requirement that it is energetic; the second problem, to get to school, is constrained only by the requirement that it take a short time. Propagating these constraints leads to the obvious solution that one should run or ride a bike to school.

Following constraint formulation and propagation, MOLGEN seeks to *satisfy* constraints. In the domain of molecular genetics, this often involves replacing an abstract object with a particular one that satisfies the constraints put on it. For example, it may involve replacing the object *bacterium* with *e. coli* and replacing the object *antibiotic* with *tetracycline*. Whatever the results of constraint satisfaction, it is facilitated by constraint formulation and propagation, which together narrow down the space of refinements that is considered for each subproblem.

The formulation-propagation-satisfaction cycle is a *constructive* process; abstract parts of plans usually are refined only when there are constraints specifying the refinement. The antithesis of this constructive cycle is found in rare cases in which MOLGEN lacks the constraints needed to refine a plan step. It guesses a refinement that may be shown at a later time to interfere with other parts of the plan, in which case the refinement is abandoned for another. This process is destructive, since it may involve throwing away old planning decisions.

An Example

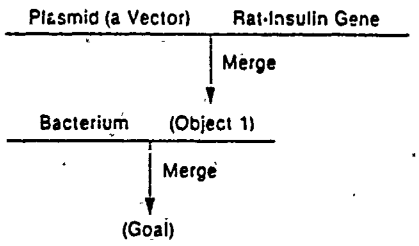
MOLGEN has been used to find plans for the *rat-insulin* experiment (Stefik, 1980). Many organisms produce insulin that is biologically active in humans but can sometimes cause allergic reactions. An alternative to extracting insulin from the pancreas of animals is to design a bacterium that produces insulin. No bacteria are known to produce insulin naturally, so one must be created. To do this, the gene coding for insulin production in rats was spliced into bacteria, altering the genetic makeup of the bacteria and causing them to produce insulin. This experiment was done in 1977; it was selected as a test case for MOLGEN, which successfully designed four different plans for the experiment.

The major steps in the experiment involved finding a medium in which to embed the insulin gene, allowing some bacteria to absorb this medium, killing off the bacteria that did not absorb the medium, and growing the culture of those that did. The plan is simple at this abstract level—that is the advantage of hierarchical planning. The complete plan is actually quite complicated and involves many constraints.

MOLGEN represents the goal of the experiment using the most abstract objects it knows of. The goal is to obtain a culture with

```
ORGANISMS = (A Bacterium with
                 EXOSOMES = (A Vector with
                                 GENES = (RAT-INSULIN))).
```

Planning in MOLGEN is driven by *means-ends analysis*, which is to say that, at each step of the planning process, MOLGEN seeks operators that will reduce the *differences* between the current state of the plan and its goal. In this case, MOLGEN makes a very abstract plan to build, from available objects, the organism specified in the goal. It plans two *merges* of objects to achieve its goal. The first merge involves the insulin gene and a *vector* (a medium for carrying the gene into the body of a bacterium), and the second merge involves the results of the first merge and the bacterium:



Next, MOLGEN refines the two abstract merges to more specific operations. The second merge, by which a bacterium absorbs a plasmid carrying new genes, corresponds to a laboratory step called a *transformation.* But MOLGEN knows that not all plasmids are absorbed by all bacteria, so it formulates the constraint that they be compatible. MOLGEN also knows that *transformation* operators work by mixing plasmids and bacteria together in a solution and that some bacteria will not absorb the plasmid. This leads to a difference between the goal of the experiment and the state resulting from the plan: The goal is to have a single culture of bacteria carrying a particular gene, but the plan results in a culture of bacteria in which some bacteria do not carry the gene.

Since planning is driven by differences between the current state and the goal, MOLGEN tries to solve the problem of getting rid of the unwanted bacteria. To do this, it proposes to *sort* the culture. *Sort* is an abstract operator that is next refined to *screening* the bacteria with an antibiotic. Note that the antibiotic is not specified because the bacterium is not. However, the refinement of *sort* to *screen* results in two constraints: that the bacteria that absorb the plasmid should resist the antibiotic and that the bacteria that do not absorb the plasmid should perish from the antibiotic.

At this point, MOLGEN *propagates* the constraints about antibiotic resistance. The result of the propagation is that both constraints on the bacteria are replaced by a single constraint on the plasmid itself. The reasoning is that, since the only difference between the two kinds of bacteria is that one carries the plasmid, the plasmid itself must confer immunity to the antibiotic. Notice that this reasoning does not change any of the plan steps that have already taken place, but it does constrain MOLGEN to include a resistance gene for an antibiotic in the plasmid.

So far, MOLGEN has done a little bit of planning at an abstract level and a lot of reasoning about how to refine the abstract plan into a detailed one. It has proposed a *merge* of a gene and a plasmid, a *transformation* of that result into two bacteria, and a *screening* of the bacteria to obtain the desired one. The identities of the bacteria, the screening agent, the resistance gene, and the plasmid that will carry the genes are unknown, but MOLGEN knows some things about these objects in the form of constraints. For example, the resistance gene and the antibiotic must be compatible, and the plasmid must be compatible with the bacterium. As MOLGEN continues to plan, particularly to plan how to insert the desired genes in a plasmid, other constraints will be formulated.

Eventually, MOLGEN will be able to satisfy constraints. By then, it will have refined the plan to a point where the only bacterium that it knows will satisfy all the constraints is *e. coli*. Similarly, it will have found just one method of inserting genes into a plasmid (though this was not done through constraint propagation but because MOLGEN knows of only one such method). It will have found two antibiotics—*tetracycline* and *ampicillin*—and four plasmids that satisfy the constraints. Thus, it finds four solutions to the rat-insulin problem.

MOLGEN's solution to the rat-insulin experiment was more complex than the abbreviated version presented here. In all, a dozen constraints emerged during the planning process. The development of the plan was complex, requiring about 30 pages of printout to document.

Conclusion

We have seen that MOLGEN can develop a complex plan without ever undoing a planning decision. Its *least-commitment* strategy dictates that it defer decisions for which it lacks constraints, and, thus, it rarely commits itself to a decision that it must later undo.

MOLGEN plans at different levels of abstraction, and it also works at three levels of planning actions to accomplish *meta-planning:* At the highest level it makes *strategy* decisions, at the middle level it makes *design* decisions, and at the lowest level it decides how to *instantiate* its design.

References

Stefik's MOLGEN system is discussed in his doctoral dissertation (1980).

E. REFINEMENT OF SKELETAL PLANS

ONE WAY to develop methods for AI systems is to observe the methods that humans use. Such an approach is typically taken by cognitive scientists (see Chap. XI) to develop models of cognition. This article describes a molecular genetics (MOLGEN) planning system developed by Peter Friedland after studying human experiment-planning behavior. The major observation of the study was that scientists rarely invent from scratch the plan for an experiment. Most often, they begin with an abstract, or *skeletal*, plan that contains the basic steps. Then they instantiate each of the plan steps by a method that will work within the environment of the particular problem. Skeletal plans range from general to specific, depending on the experimenter and the problem. This MOLGEN system is one of two such systems developed at Stanford University; the other, by Mark Stefik, is discussed in Article XV.D2.

This article gives an example of skeletal plans in the laboratory and discusses the implementation of the method in the MOLGEN system for the design of experiments in molecular biology.

Two Examples of Analysis Experiments

As an introduction to the skeletal-plan method, two simplified and related examples of analysis experiments in molecular biology are presented, namely, DNA sequencing and restriction-site mapping. Both experiments involve similar sequences of actions; consequently, they are discussed as variants of a single skeletal plan.

DNA sequencing: The problem. DNA is composed of a linear string of molecules called bases. There are four possible bases, adenine, cytosine, guanine, and thymine, usually abbreviated A, C, G, and T. The goal of a sequencing experiment is to determine which of the four bases is present at each position on the molecule. The base sequence is extremely important in determining both the biological function and the physical structure of the entire DNA molecule.

DNA sequencing: The solution. One of the best current experimental plans for DNA sequencing, known as Maxam-Gilbert sequencing (Maxam and Gilbert, 1974), is as follows:

1. Label one end of the molecule with radioactive phosphorus. This gives the experimenter a "handle" for later locating pieces of the molecule attached to the radioactive end. Radioactive phosphorus labeling is the current method of choice for end-labeling of DNA.

2. Divide the sample into four portions. For each portion, apply a hydrazine-based chemical reaction that cuts the molecule at a particular base. Control the reaction so that, on the average, one base is cut per molecule. Each of the four samples will then contain a population of molecules of lengths determined by the base that was cut in that sample.

3. Determine the lengths of the molecules in each population with a labeled end. This is done by a technique called acrylamide gel electrophoresis, which is currently the most accurate method for the separation of molecules by length.

For example, suppose the starting sequence was AGTTCGA. The sample for which the molecule was cut at the A base would show labeled molecules of lengths 0 and 6, the C sample would show molecules of length 4, the G sample would show molecules of lengths 1 and 5, and the T sample would show molecules of lengths 2 and 3. The sequence can now be "read" directly from these lengths.

Restriction-site mapping: The problem. Restriction enzymes are used to cut DNA molecules at specific locations. The locations are specified by a pattern of four, five, or six bases called a restriction site. The goal of a mapping experiment is to find all of the restriction sites for common enzymes on a molecule. This information tells the molecular geneticist which enzymes to use or not to use in a future experiment that requires restriction cutting.

Restriction-site mapping: The solution. One of the best current methods (Smith and Birnsteil, 1976) is as follows:

1. Label the end with radioactive phosphorus as above.

2. Divide the sample into as many new samples as restriction enzymes for which a map is desired. Then, for each sample, do a "partial digest" with one restriction enzyme. This means to control the laboratory conditions (temperature, pH, time of application) so that only one or two sites are cut on the average molecule. As above, a population of molecules will exist in each sample.

3. Determine the length of the labeled molecules by means of electrophoresis, as above. The length measurements will locate each of the restriction sites for each enzyme tested.

The Skeletal Plan Refinement Method

Clearly, the two experiments described above are closely related. Each had the goal of locating the position of a specific site—either a single base or ~~a string of bases—on the molecule. Each had the same design; they differed~~ only in the middle, cutting step. Both experiments sprang from the same basic idea:

1. Label one end of the molecule:
2. Cut with an agent that makes an average of one cut per molecule at the sites that are being mapped;
3. Determine the length of the labeled fragments.

This is an abstracted or skeletal plan that is useful for locating any type of site for which there is a suitable cutting agent.

The plan is transformed into an actual design for an experiment by refining each step in the plan—by instantiating it with a method that will actually work in the laboratory. The first and third steps of the experiments—phosphorus labeling and gel electrophoresis—were chosen because they were the best methods available. The choice of the second step was directed by the specific choice of site to be mapped.

The idea here, again, is that scientists rarely invent an experimental design from scratch. They find a strategy, a skeletal plan, that was useful for some related experimental goal and then instantiate it with the proper laboratory methods for their specific goal and laboratory conditions. The skeletal plan may be very specific if the goal is similar to one for which a very good experiment has already been designed. It may also be extremely general, as was the plan in the example above.

Implementation in MOLGEN

The skeletal plan method is used successfully in the MOLGEN system. Since the method depends heavily on domain knowledge, a well-organized, expert knowledge base is the central part of the system. The Unit package (Stefik, 1979) is used by domain experts to construct a knowledge base containing both a selection of skeletal plans and the objective and procedural knowledge necessary to instantiate the plans competently. The Unit package permits the domain experts to describe such information in a language natural to them as molecular biologists.

The two major steps in planning by incremental refinement of skeletal plans—plan selection and plan-step refinement—are described separately below.

Choosing a skeletal plan. Skeletal plans are specified at many levels of generality. At the most general level, there are only a few basic plans. These are used as fallbacks when plans that are easier to refine and that are more specific cannot be found. The problem is not just one of finding a plan that might provide a satisfactory solution, but of finding a plan that will require the least refinement work. Skeletal plan finding reduces to a simple lookup when exactly the same problem has been solved before (even if this were done with a completely different set of laboratory and molecular conditions), but it becomes more difficult when only related problems have been solved. Then the task may be to decide whether to choose a detailed plan for a related problem or to choose a more general plan for a class of problems.

The MOLGEN work has only begun to treat these problems of plan selection. Plans are classified according to their perceived utility by molecular geneticists. The specificity of the utilities (any given skeletal plan could have many) is totally up to the experts. The knowledge base contains also a taxonomy of goals in molecular biology. When a problem is described to the planning system, a search is made of the skeletal-plan utilities to see if any exactly match the experimental design goal. If several do. all are tried; if none does, a more general goal is chosen from the taxonomy and the process is repeated.

Refining the skeletal plan. Refinement of the skeletal plan is the process of selecting an appropriate ground-level instantiation for each step in the abstract plan. In the example above, the ground-level instantiation of labeling was radioactive phosphorus. This refinement process is usually hierarchical; a scientist might decide first on the method of cutting, then on a cutting enzyme, and finally on a specific enzyme.

Knowledge about laboratory techniques is organized hierarchically in MOLGEN. There were several broad classes of techniques, with as many subclasses as are deemed natural by the domain experts. In all, about 400 different techniques are described in the knowledge base.

The MOLGEN system proceeds linearly through the steps of a selected skeletal plan. The steps are matched to the techniques in the knowledge base by name, synonym, or function of the step. A specific technique—as specific as can be directly determined from the plan step—is chosen, and then the instantiation process begins.

The knowledge to do the instantiation is stored in the form of an English-like procedural language within the knowledge base. This knowledge represents three major criteria for plan-step instantiation. In order of priority of application they are:

1. Will the technique, if successfully applied, carry out the specific goal of the step; for example, will a separatory method not just do some sort of separation, but also separate all circular DNA from all linear DNA?

2. If the technique satisfies the first criterion, can it be successfully applied to the given molecule under the given laboratory conditions?

3. Is the technique the "best" of those that passed the first two tests? This choice point, while in some sense the least important (since all techniques that make it to this point will work), seems to be the hardest for scientists to define. All the scientists studied gave somewhat different metrics involving reliability, convenience, accuracy, cost, and time to carry out the technique. The heuristic chosen as most representative gave greatest weight to four-point scales of convenience and reliability as an initial filter.

This knowledge is used to proceed down a level in the technique hierarchy; the process is repeated until an actual instance of a technique is chosen. At higher levels of the hierarchy (i.e., with less refined plans), a premium is set on achieving goals; but at lower levels of the hierarchy, a premium is set on making plans efficient and elegant.

This strategy-finding process is common to many disciplines. In his book *How to Solve It*, Polya (1957) describes "mobilizing" problem-solving knowledge:

> Many of these questions and suggestions aim directly at *mobilization* of our formerly acquired knowledge: *Have you seen it before? Or have you seen the same problem in a slightly different form? Do you know a related problem? Do you know a theorem that could be useful?* (p. 159; italics in original)

The idea is to avoid reinventing general strategies and to use plan outlines that have worked in the past on related problems.

Related Work

The concept of a skeletal plan for experimental design has a direct precedent in Schank and Abelson's work in natural-language understanding (see Article IV.F6, in Vol. I). They introduce *scripts*, declarative representations of ordered sequences of events. The detailed knowledge contained in scripts is used to understand, predict, and participate in events one has encountered previously.

Schank and Abelson also introduce generalized scripts, called *plans*, that explain events related to, but not exactly like, those the user has seen before. "Plans are where scripts come from.... The difference is that scripts are specific and plans are general" (Schank and Abelson. 1977, p. 72). In fact, there is a continuum between scripts and plans in Schank and Abelson's work: "There is a fine line between the point where scripts leave off and plans begin.... When a script is available for satisfying a goal, it is chosen. Otherwise a plan is chosen" (p. 77; see also Article IV.F6, in Vol. I).

The idea of abstracted plans is found also in the STRIPS planner (Fikes, Hart, and Nilsson, 1972; see also Article II.D5, in Vol. I). This system parameterized successful plans in order to generalize them. The generalized plans were called MACROPs (for macro-operators).

There are several distinctions between skeletal plan refinement and some of the other methods discussed in this chapter—for example, Stefik's parallel work on planning in molecular biology (see Article XV.D2). Other methods emphasize building the initial abstract plan; this method assumes the initial plan is already known and emphasizes the plan selection and instantiation process. ~~Other methods concentrate on the interaction of plan steps; this~~ method, in large part, considers plan steps to be sufficiently independent that conflicts can be resolved by relatively minor subplans. Finally, other methods place relatively little emphasis on domain-specific expertise, whereas such expertise is the heart of this planning method.

Conclusion

The reader may be surprised by the simplicity of the method of skeletal plan refinement but should remember that it attempts to produce competent—rather than wildly innovative—plans. It is based on the observation that human scientists who know a lot about their domains, and who have flexible cross-associations for choosing steps in an experiment, are usually good at experimental design. There are very few totally new plan outlines discovered, but many new plan instantiations.

References

A source for this article and a good discussion of this implementation of MOLGEN is Friedland's doctoral dissertation (1979).

BIBLIOGRAPHY

Bobrow, D. G., and Raphael, B. 1974. New programming languages for artificial intelligence. *Computing Surveys* 6.

Corkill, D. D. 1979. Hierarchical planning in a distributed environment. *IJCAI 6,* 168–175.

Feitelson, J., and Stefik, M. 1977. A case study of the reasoning in a genetics experiment. Rep. No. HPP-77-18, Heuristic Programming Project, Computer Science Dept., Stanford University.

Fikes, R. E., Hart, P. E., and Nilsson, N. J. 1972. Learning and executing generalized robot plans. *Artificial Intelligence* 3:251–288.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189–208.

Friedland, P. E. 1979. Knowledge-based experiment design in molecular genetics. Rep. No. 79-771, Computer Science Dept., Stanford University. (Doctoral dissertation.)

Hayes-Roth, B. 1980. Human planning processes. Rep. No. R-2670-ONR, Rand Corp., Santa Monica, Calif.

Maxam, A., and Gilbert, W. 1974. A new method for sequencing DNA. *Proceedings of the National Academy of Sciences* 74(2):560–564.

Miller, G. A., Galanter, E., and Pribram, K. H. 1960. *Plans and the structure of behavior.* New York: Holt.

Newell, A., and Simon, H. A. 1972. *Human problem solving.* Englewood Cliffs, N.J.: Prentice-Hall.

Polya, G. 1957. *How to solve it.* New York: Doubleday Anchor Books.

Sacerdoti, E. D. 1974. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence* 5:115–135.

Sacerdoti, E. D. 1975. A structure for plans and behavior. Tech. Note 109, AI Center, SRI International, Inc., Menlo Park, Calif. (Doctoral dissertation.)

Sacerdoti, E. D. 1979. Problem solving tactics. Tech. Note 189, SRI International, Inc., Menlo Park, Calif.

Schank, R. C., and Abelson, R. P. 1977. *Scripts, plans, goals, and understanding.* Hillsdale, N.J.: Lawrence Erlbaum.

Smith, W., and Birnstein, M. 1976. A simple method for DNA restriction site mapping. *Nucleic Acids Research* 3:2387–2398.

Stefik, M. J. 1979. An examination of a frame-structured representation system. *IJCAI 6,* 845–852.

Stefik, M. J. 1980. Planning with constraints. Rep. No. 80-784, Computer Science Dept., Stanford University. (Doctoral dissertation.)

Sussman, G. J. 1973. A computational model of skill acquisition. AI Tech. Rep. 297, AI Laboratory, Massachusetts Institute of Technology. (Doctoral dissertation.)

Sussman, G. J. 1975. *A computer model of skill acquisition.* New York: American Elsevier.

Tate, A. 1975a. Interacting goals and their use. *IJCAI 4*, 215–218.

Tate, A. 1975b. *Using goal structure to direct search in a problem solver.* Doctoral dissertation, University of Edinburgh.

Tate, A. 1976. Project planning using a hierarchic non linear planner. Rep. No. 25, AI Research Dept., University of Edinburgh.

Waldinger, R. 1977. Achieving several goals simultaneously. In E. W. Elcock and D. Michie (Eds.), *Machine intelligence 8.* New York: Halstead/Wiley.

NAME INDEX

Pages on which an author's work is discussed are italicized.

SUBJECT INDEX