MEMORANDUM
RM-3283-PR
DECEMBER 1962

SOME PROBLEMS OF BASIC ORGANIZATION IN PROBLEM-SOLVING PROGRAMS

Allen Newell

PREPARED FOR:
UNITED STATES AIR FORCE PROJECT RAND

The RAND Corporation
SANTA MONICA • CALIFORNIA

MEMORANDUM
RM-3283-PR
DECEMBER 1962

SOME PROBLEMS OF BASIC ORGANIZATION IN PROBLEM-SOLVING PROGRAMS

Allen Newell

This research is sponsored by the United States Air Force under Project RAND — Contract No. AF 49(638)-700 — monitored by the Directorate of Development Planning, Deputy Chief of Staff, Research and Technology, Hq USAF. Views or conclusions contained in this Memorandum should not be interpreted as representing the official opinion or policy of the United States Air Force. Permission to quote from or reproduce portions of this Memorandum must be obtained from The RAND Corporation.

The RAND Corporation
1700 MAIN ST • SANTA MONICA • CALIFORNIA

PREFACE

This Memorandum is an interim report on the development of problem-solving computer programs. Several of the organizational problems in constructing a problem-solving program and applying it to actual problems are discussed and analyzed in an effort to detect the sources of the information contained in the program.

The study is of interest to programmers and engineers concerned with advanced concepts of machine use and machine organization, and will eventually lead to construction of very sophisticated applied programs, patterned on the game-playing and theorem proving type programs common today.

A presentation based on this study was made at the Conference on Self-Organizing Systems, Chicago, Illinois, May 1962, sponsored by the Information Systems Branch of the Office of Naval Research and the Armour Research Foundation of the Illinois Institute of Technology.

SUMMARY

This Memorandum examines several examples of organizational problems dealing with the construction and application of problem-solving programs. Usually, problem-solving programs are discussed by giving only the results and the main methods used to achieve these results. Yet, most of the difficulties in creating programs stem from organizational problems, not substantive ones. This Memorandum is an attempt to bring some of these out in the open. It proceeds entirely by examples drawn from the experience of the last several years, since no adequate theoretical or conceptual framework exists for discussing organizational issues.

The first example is how to store information that is created dynamically and unpredictably during the operation of the program. The solution that has been adopted--list processing--is already well known, but is reviewed to bring out its essential features.

The second example is how to organize large, complex processes. The two principles that are at the heart of modern programming, sequential control and hierarchical subroutine organization, are discussed, both as to their power and their limitations. The latter show up in the form of highly rigid programs.

The third example is how to have many different kinds of goals producing many different kinds of results, and yet be able to use these results in the rest of the problem.

The problem is made more difficult when it is also desired to create and attempt goals in arbitrary order. Early problem-solving programs solved the difficulty by establishing restrictive conventions on the types of goals allowed. Some attempts to remove these restrictions that have been tried with General Problem-Solver (GPS) are discussed.

The fourth example is how to avoid the rigidities of many special routines when building up highly particular and inhomogeneous collections of data. This is a problem in which list processing is only of modest help.

The last example concerns the general problem of how to remember the past. A problem-solver must selectively keep various amounts of information about its past history. Rigid strategies for doing this lead to cumbersome programs and it is not clear how to provide the necessary flexibility.

All of these problems stem from the fact that problem-solving programs are more dynamic and require more flexibility than we know how to provide. By solving these organizational problems in this context we can expect to develop the appropriate ways to organize complex programs that require flexibility in many applied areas as well.

CONTENTS

I. INTRODUCTION

There now exists a well-launched scientific enterprise to explore the information processes involved in intelligent behavior by constructing computer programs that perform in an intelligent way. One segment of this enterprise focuses on programs able to solve difficult problems. About a dozen programs have been built that can fairly be called problem-solvers. They include checker playing programs;[1,2] chess playing programs;[3,4,5] theorem proving programs in symbolic logic[6] and plane geometry;[7] programs for handling management science problems;[8] programs for analytical integration;[9] some attempts at more general problem solving programs;[10] plus a few others.

These programs are all rather similar in nature. For each, the task is difficult enough to allow us to assert that the programs problem-solve, rather than simply carry out the steps of a procedure for a solution invented by the programmer. They all operate on formalized tasks, which, although difficult, are not unstructured. All the programs use the same conceptual approach: they interpret the problem as combinatorial, involving the discovery of the right sequence of operations out of a set of possible sequences. All the programs generate some sort of tree of possibilities to gradually explore the possible sequences. The set of all sequences is much

too large to generate and examine in toto, so that various devices, called heuristics, are employed to narrow the range of possibilities to a set that can be handled within the available limits of processing effort. Within these bounds there is a good deal of variation among the programs as to the particular heuristic devices used. These range from learning schemes,[1] to models,[7] to elaborate abstract representations.[8] The current state is well summarized by Minsky.[11]

For all the simplicity of the thumbnail sketch given above, these programs are large and complex. Each has required large amounts of effort to develop. Since these programs represent an attempt to get computers to perform patterns of symbolic activity that we do not understand well, their complexity and effort is not surprising. We do not know whether the difficulties stem from our ignorance or from the inherent complexity of the processing.

It is usual, when talking or writing about these programs, to describe the task the program is to accomplish, the major methods and heuristics that are used, and the top executive routine. Problems in program organization--what system conventions were adopted and how they affected the problem, or what role different data structures played--are not much discussed. (See (10) for a typical example.) This silence is not surprising. The performance

of the program comes first. Without some fairly interesting problem-solving behavior, of what interest are organizational problems? More important, we neither know what to discuss about organization, nor how to discuss it. There is no lack of appreciation of the problems. Any systems programmer can testify that questions of representation, communication conventions, and the like, are the bane of his existence--and the reason for it.

For problem-solving programs, the one exception to this dearth of attention to organization is the development of list structures and list processing languages. Here an organizational problem pressed hard enough to call forth an extended response and we found a way of talking about it--partly, I suspect, because the solution was expressed as a language. The literature on list processing is appreciable[12-21] and where discussions of organizational problems do occur they usually center around list processing languages. [8, Chapter 6] A programming language is a way of dividing up organizational problems. The language encapsulates the solutions to one set of problems--for list languages these solutions are new data representations, ease of hierarchization, recursive programming, etc. It leaves another set untouched--how to build large programs in terms of the language. Constructing an Information Processing Language (IPL) or a List Processor (LISP) does not automatically produce a problem-solver.

The purpose of this paper is to discuss some organizational aspects of problem-solving programs. Of necessity it will proceed by a sequence of examples. Although several issues can be identified--such as centralization versus decentralization--there is no adequate framework in which to discuss them generally and abstractly. Each example must be described in particular detail and in its own terms.

Each example revolves around a specific issue. Most of them have been drawn from programs with which I am intimately familiar; namely, Logic Theorist (LT),[6] our chess program,[5] and especially GPS,[10] although I have used the experience from other programs where it has been known to me. No claim is made that these organizational problems form an exhaustive list, or are representative or original. However, they have commanded attention at one time or another. Finally, most of them are unsolved to some extent, either completely, or because the solutions that have been adopted are still unsatisfactory in one way or another.

II. STORING DYNAMIC STRUCTURES

Let us start with a simple case that is already well known and relatively successfully solved. The problem is to find memory space for the expanding tree of subproblems that is generated in exploring for a solution to the main problem. In chess, for example, one starts with the current position; analysis indicates certain moves as worth considering and the positions resulting from these moves are generated. Analysis is again conducted from these new positions, yielding additional moves and additional positions. Thus the tree of information, shown at one instant in Figure 1, grows during the course of problem-solving. The amount of information stored for each subproblem is indicated by the length of the bar at each node; it is variable since the description at a position depends on the analysis. The problem, then, is how to allocate space in a standard computer memory for this growing tree.

When this problem first arose,[18] the existing techniques of memory allocation revolved around the assignment of individual cells of memory for numbers and continuous segments of memory for vectors, matrices, and tables. Arbitrary symbols for addresses could be used in programming, the actual assignment of machine addresses being deferred to the time just preceding execution when

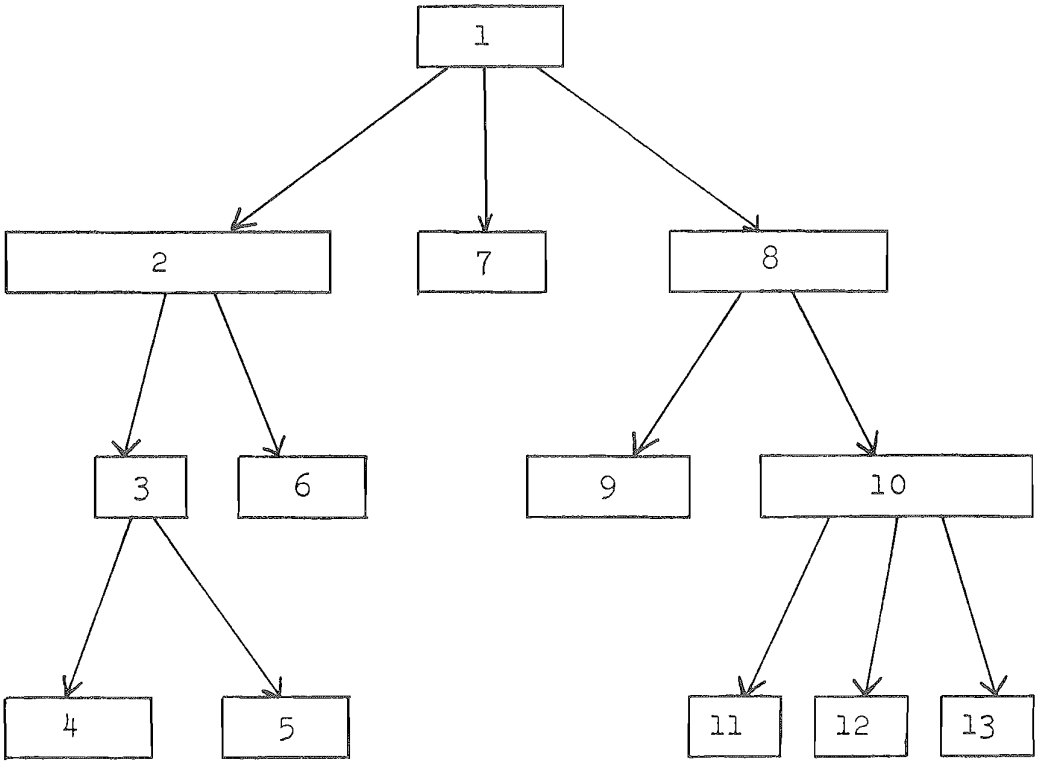


Fig. 1. Growing Tree Structure.

all entities demanding space have been specified. No techniques were at hand for the dynamic case in which the entities requiring space come into existence while the program is being executed.

The solution to this problem is well known: the organization of data in list structures. The essential ideas are three. The first is to eliminate the topological structure defined by the sequence of machine addresses and replace it by explicit links. When this is done, a list of consecutive items can be distributed anywhere through memory, and new items can be added or deleted from any part of a structure without disturbing items that already exist. The second idea is to put all the memory cells not in use on a single list with a known name, the available space list. This list becomes the source of new cells, when they are required, and the repository for cells no longer needed. The final idea is the creation of a set of processes for manipulating list structures--processes such as "insert," "delete," "find last," "erase," and so on.*

---

*Several variants of list processing already exist. They differ in the occasions and the assignment of responsibility for returning cells to available space; in the size and variability of the units that are linked; and in the kinds of access to list structures that are provided. None of these variations affects appreciably the solution provided for the dynamic allocation of memory.

Several of the problem-solvers use list techniques [5-10] but several do not,[1,2,3] and it is interesting to understand the reason. The crucial difficulties arise only when parts of the tree must be erased to gain space for new parts. In general this provides an irregular distribution of odd-sized pieces of space with which to meet the additional demands for space. However, special strategies for processing the tree can be found that avoid these difficulties. The one used by the problem-solvers can be called the "depth first" strategy. It is indicated by the following recursive formula:

```
Process node X
    For each immediate subnode of X (call it Y):
        Create node Y
        Process node Y
        Save summary information from Y
        Erase node Y
    Finish the analysis of node X.
```

The nodes of Figure 1 are numbered according to the order of generation specified by this strategy. When this strategy is used, the only nodes that need to be in existence at any moment are those that extend from the top down a particular path towards the bottom. Assignment of space always proceeds from the righthand edge of the occupied space into open space, and erasure always occurs from the right, enlarging the open interval. Thus this strategy avoids most of the problems of dynamic assignment by imposing some very special restrictions on the

processing: nodes are considered only once and in a fixed order. Happily, the minimaxing procedure used in most game programs is compatible with these restrictions. More flexible exploration of the tree requires a more general system of storing information.

Can we describe in abstract terms what is involved here? We started out with a set of techniques that required certain kinds of structure to be fixed and known to the program. The problem-solving programs required that memory assignment be variable. A successful solution has been achieved, not just by building programs that would detect the variability and react accordingly, but by creating a new set of invariant concepts (the list processes) so that the variability was taken care of without an increase in program complexity.

III. ORGANIZING LARGE PROCESSES

The processes that accomplish problem-solving are large and complex. How can we specify such processes? This seems almost a misplaced question. The computing art has provided us with two tools for synthesizing complex processes--the sequential flow of control, and the hierarchy of closed subroutines--and it was in terms of these tools that we even conceived the possibility of constructing problem-solvers. But these tools, however automatic their adoption, carry with them implications for what is easy and what is hard--for what capabilities will be included and what will be left out.

Sequential processing is, of course, built into the basic structure of our machines. But it extends much further than this, being used at the most macroscopic level in the programmer's flow diagram. It encourages us to envision isolated processes devoted to specific functions, each passively waiting in line to operate when its turn comes. It permits us to think of the total program in terms of only one thing going on at a time.

The subroutine hierarchy emphasizes even more strongly the notion of isolated and well-controlled processes. Each subroutine has its well-specified inputs and provides its well-specified outputs. The inputs and outputs define all it must do and the sum of its interactions with the

total processes. The situation is perhaps more vivid when viewed from the other side: when a routine uses a subroutine, it need know only the input-output specifications of the latter. The using routine may safely ignore whatever complexity and involved processing goes on inside the subroutine. There is none of the Alice-in-Wonderland croquet game, with porcupine balls unrolling themselves and wandering off, and flamingo mallets asking questions at inopportune times.

I don't mean to undervalue the importance of sequential control and closed subroutines as organizing principles. They constitute an organization theorist's dream. By isolating each separate task, they allow us to think through each part of our program in relative security, knowing that there will be few interactions with other tasks, and that we can depend on each part playing the role assigned to it. These organizing principles have solved several other problems as well. They allow nearly identical subprocesses to be coded once and for all as single subroutines with parameters that can be changed; the savings in coding effort and in errors are worth almost more than the savings in space. They aid in debugging, by permitting individual parts of the program to be debugged separately. Perhaps most important, they aid in program modification, where centralization of processing has turned out to be crucial.

Many big changes in a program that one would like to make and can't are thwarted because some crucial process is distributed through the program and handled in idiosyncratic ways, calling for innumerable and difficult corrections to make the change.

Indeed, so powerful is the concept of the sequentially controlled hierarchy for organizing large processes, that some feel that the extremely convenient techniques for subroutinization provided by the list languages are their biggest asset.(14) But there are difficulties. They stem from two related effects and can be summarized in one word: rigidity.

The first difficulty is that this kind of organization calls for uniform conventions to specify how one subroutine will communicate with another. These conventions take such forms as the calling sequence of standard programming usage, the communication list of IPL, and the functional notation (i.e., F(X,Y,Z)) of algebraic languages and LISP. Although these conventions may carry no such implication in principle (an irrelevancy), in practice they lead to minimizing the amount of communication between routine and subroutines. In consequence, the subprocesses are forced to work in an impoverished informational environment. Processes that do large amounts of work on small amounts of data tend to be preferred by the programmer to processes that use fragments of many

different kinds of data. Yet the latter kind seems to fit better the requirements of problem-solving, in which relatively weak and scattered information must be used to guide the exploration for a solution.

The difficulty might be alleviated by maintaining the isolation of routines, but allowing all the sub-routines to make use of a common data structure. Metaphorically* we can think of a set of workers, all looking at the same blackboard: each is able to read everything that is on it, and to judge when he has something worthwhile to add to it. This conception is just that of Selfridge's Pandemonium:[22] a set of demons, each independently looking at the total situation and shrieking in proportion to what they see that fits their natures. It is a much easier mode of organization to use when the processes are only perceptual--that is, look at, but do not change, the environment--than with active problem-solving where the workers are as busy writing and erasing as they are reading. Thus many large programs, especially command and control programs, do have common pools of information which play exactly the role described here, but these programs are highly restrictive as to the interactions they allow.

---

*Metaphors, especially those involving human organization and human activities, provide highly appropriate guides for machine organization. We shall use them freely.

Considerations of this kind has a strong influence on the design of the later IPL's. IPL-II, the list language used to program LT,[18] had a centralized way of taking input information from the routine at the higher level and establishing it in working cells for the next lower level. This scheme enforces the communication of all immediate context information via the input sequence. Information often has to pass through many levels to get from the routine that generates it to the routine that ultimately uses it. Such information is still "local" in that it is created internally by some routine in the course of doing its job; it cannot easily be considered part of the absolute context and be given a universal name.

Beginning with IPL-III[16] we have adopted the push-down list as a technique for avoiding this rigidity in communication among processes, and encouraging a more "blackboard" kind of operation. Each cell individually may be pushed-down to save the information being used by a higher routine, and popped-up to return it to use. Imagine, then, an array of cells holding information for an hierarchy of subroutines. Communication between routines occurs by means of these cells, each cell holding information of a specific kind so that the routines know where to find information they need and put information they produce. The information in all the cells is available to all the routines in the hierarchy. Where it is

necessary to communicate to a subroutine some information in a cell that is different from the current information, then the cell is pushed-down, the new information put into it, the subroutine executed, and the cell popped-up after the subroutine is finished. This sequence does constitute a deliberate act of communication, as definite and expensive as the standard procedures. But it occurs only on those cells that need to be changed; the information in the remaining cells is automatically communicated. This system operates on a principle of exceptions, whereas current subroutine communication philosophy dictates that everything must be actively communicated.

Although it is possible to show some positive benefits from the individualized push-down list philosophy, it has not changed the character of the large programs in major respects. They still have fundamentally the flavor of a hierarchy with restricted communication. Clearly, this device did not tackle the right difficulty.

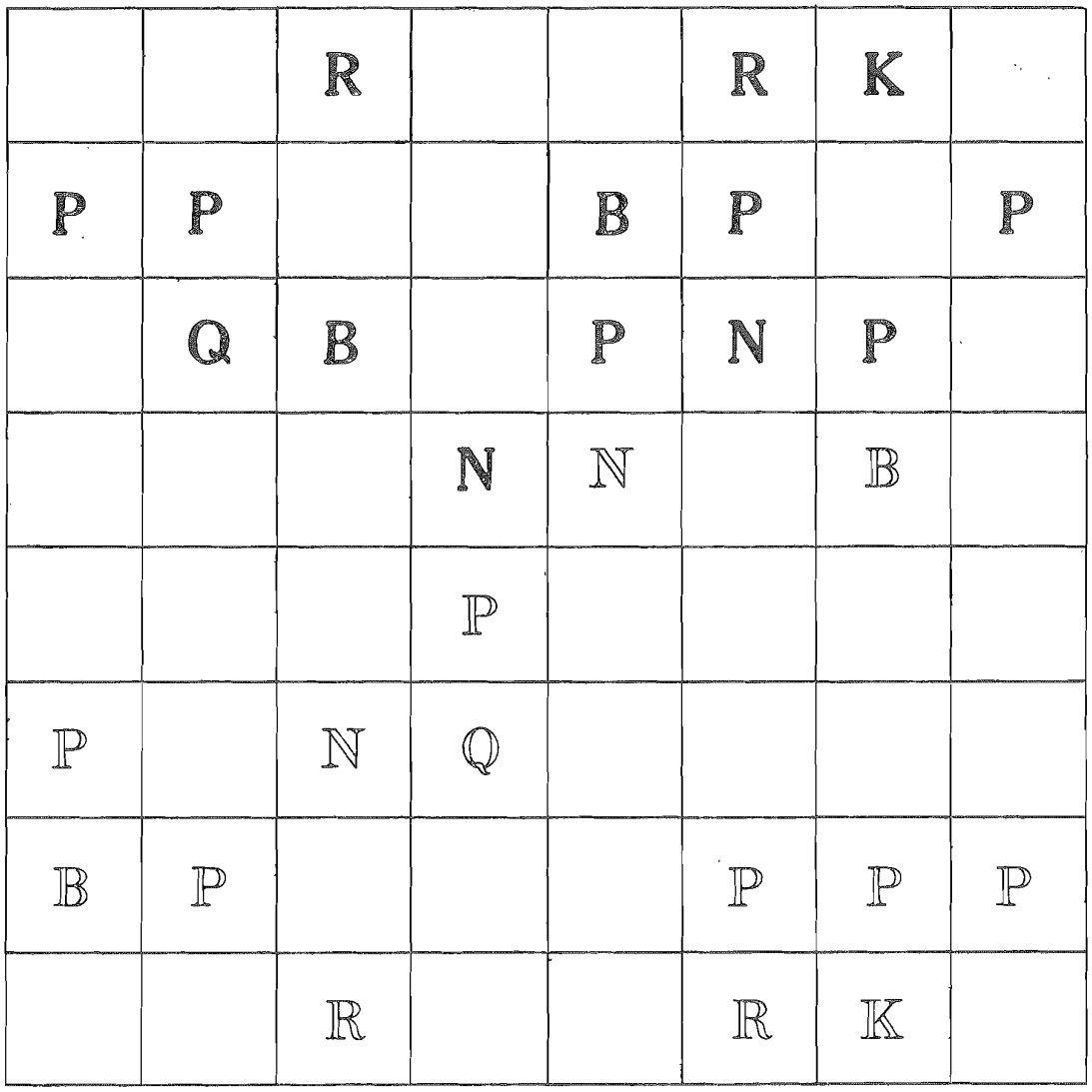
A second problem of rigidity associated with organizing large processes in a sequentially controlled hierarchy relates to high level organization. The subroutine and the rigid input-output relation derives from and fits very well the mathematical concept of function. At a low level in a program--at the level of sines and cosines, or of inserts and deletes--one wants highly specific tools that behave in an exact, prescribed fashion. At higher

levels, this rigidity has serious disadvantages. Consider an example.

In studying how humans play chess, we were led to consider the position shown in Figure 2, taken from de Groot's work on chess.(23, page 65) We wanted to know what our chess program would do in such a position, playing for Black. The situation is complex, but there is an obvious undefended White Pawn at KN2. Capturing this Pawn is unsound, although it is not immediately obvious. Good human players spend most of their time considering the center region of the board. As you might expect, our chess program did not see deeply enough to find the fallacy, and so took the Pawn; in fact, it explored the center relatively little.

The question now arose: could we get the program to explore the center? Could we say to it, in effect, "If you didn't take the Pawn; what would you do?" or "Why did you ignore the center? ... (at which point the program might reply by giving us its sketchy analysis of the position) ... But that isn't enough; analyze it further." It turned out there was no way to do this. We considered various subterfuges, such as removing the Pawn from the board, or assigning it zero value; we even tried one or two of these. None of the tricks worked, because of the elaborate interrelationships among the parts of the program. None of them produced an analysis

BLACK



WHITE

Fig. 2. A Chess Position from De Groot.

that was an appropriate response to the question we were trying to pose.

Should we be disturbed? Should we expect our program to be able to answer such a question? Our chess program--like all the others--is a big subroutine which inputs chess positions and outputs moves. This task pervades the entire structure of the program. All information and organization that does not contribute to this end is considered excess baggage and removed if possible. Yet in some sense the questions we asked it above are well within the basic power of analysis of the program. It is "only" a matter of organization. It seems a peculiar intelligence which can only reveal its intellectual powers in a fixed pattern.

One can think of ways to make this demand for flexibility operational. Let us stipulate that for each problem-solver there be specified an input language, such that the problem-solver can respond to any request stated in the language. We can define the breadth of a program by the range of things it can respond to, and its power by the difficulty of the problems it can solve. Certainly we can demand of it that, if it can answer question X successfully, it also be able to deal with all questions similar to X and obviously easier than X.

The value of this formulation is in focusing on the missing features of the higher organization of our

existing problem-solvers. Instead of fixed executive routines, processes are needed to interpret the incoming requests in the problem language and to organize the available parts of the program into a functioning unit for the task at hand. In these terms, Baseball,[24] the question-answering program developed by Green and his colleagues at Lincoln Laboratory, has greater breadth than any other problem-solver, although it does not have great power.

IV. USING THE RESULTS OF SUBGOALS

Our previous two examples covered familiar ground: subroutine hierarchies are pervasive in all complex programming; and list processing is the one major contribution of problem-solving programs to the programming art. Let us now move to less familiar territory.

Suppose a subgoal is attained--how does the problem-solver make use of the results? This seems a little like asking a man how he would spend a thousand dollars. Yet it should come as no surprise that difficulties occur in getting machines to do what should come naturally. For anything to happen in a machine, some processes must know* enough to make it happen. Thus, the results secured by attaining subgoals will be used only if routines exist that know how to use them. And the nature of this information--its exact content and the ways in which it becomes known--conditions the kinds of results that can be secured,

*We talk about routines "knowing." This is a paraphrase of "In this routine it can be assumed that such and such is the case." Its appropriateness stems from the way a programmer codes--setting down successive instructions in terms of what he (the programmer) knows at the time. What the programmer knows at a particular point in a routine is what the routine knows. The following dialogue gives the flavor. (Programmer A looking over the shoulder of B, who is coding up a routine.) "How come you just added Z5 to the accumulator?" "Because I want..." "No, I mean how do you know it's a number?" "All the Z's are numbers, that's the way I set it up." (B now puts down another instruction.) "How can you do that?" "Because I cleared the cell to zero up here at the start of the routine." "But the program can branch back to this point in front of you!" "Oh, you're right; I don't know it's cleared to zero at this point."

and through this the kind of goals that can be formulated. All this is well illustrated in the programs built to date.

LT, one of the earliest problem solvers, has for its goal tree a tree of logic expressions. Its methods are of the form, "To prove expression A, it is sufficient to prove expression B."* Thus the expressions are the only information that has to be remembered in the goal tree. LT's methods incorporate an extremely powerful organizational restriction. Once a subgoal (i.e., an expression) is formed, it becomes independent of the circumstances of its creation. If a subgoal is attained (an expression proved) then it is uniformly true that the original problem is solved. There is no issue of what to do with a successful subgoal--the first such goal that occurs terminates the entire problem-solving attempt. Furthermore, their homogeneity of form (all goals being expressions) allows all the goals to be put into a single pool, called the subproblem list, from which the executive can fish for subproblems to its liking. Complete freedom exists as to the order of generation of the subgoals and their selection for further exploration.

Thus LT solves the subgoal organization problem by avoiding it. The price paid is a substantive restriction

---

*LT has one other method of the form, "To prove A make it identical to a theorem." Although crucial for success, this method is not of interest here since it does not elaborate the goal tree.

on the kinds of methods that can be used. Gelernter, in building the Geometry Theory program,[7] eased these restrictions, but in a way that retains the main organizational advantages. The methods of his program are of the form, "To prove theorem A it is sufficient to prove theorems B and C and ... and Z." The proof of a subtheorem is not the end of the story; there are still others to go. All the subgoals (the generated theorems-to-prove) can still be put in a single pool, but the routines for selecting subproblems to be worked on must now be sensitive not only to the character of a theorem, but to the character of its siblings. However, the main simplicities still remain: all subgoals are of the same form; and the status of a goal expression is either 'proved' or 'unproved,' whereupon the executive knows uniformly how to draw the consequences for the supergoal.

In real life goals are of diverse character. Their attainment produces a partially modified state of affairs, which in some manner is to enter into the larger modification that is to be the result of a higher goal. In real life--and in our problem-solvers if they are to be good--there can be no rigid frame for the kinds of modifications that goal attainments represent. But all the early problem-solving programs have conventions that re-

strict the goals severely.* If these conventions of uniformity are destroyed, who will know enough about the results of a goal to use it? Since each subgoal is set up and used within the context of its supergoal, one answer is that processes should be associated with each supergoal that know how to deal with the subgoal.

The solution implied above, with its associated philosophy, was adopted in the first version of GPS. Goals could be of various types, and although there were only three at the start, new goal types were to be expected and welcomed. With each goal type was associated a set of methods. A method was a routine that either attained the goal directly, or decomposed the goal into appropriate subgoals and reintegrated the results of attaining the subgoals. For example, there was a "transform" type of goal for finding a way to get from one object to another. Associated with a transform goal was a method (the match method) that decomposed the goal into two subgoals. The first was a "reduce" type goal for eliminating a perceived difference between the two objects; the second was another transform goal for changing the object produced by the

---

*For instance, game playing programs are all structured so that the result of a subgoal (a game position) is a "value" of uniform character. The values of all the subgoals to a goal are combined by the executive according to the minimax rule to yield the value of the goal.

reduce goal to the final desired object. These two subgoals were of different types and produced different results. Reduce goals produced new objects; transform goals produced sequences of operators, i.e., ways to get between objects. These products were handled in quite different ways. In the example just given, the new object was used by the match method to construct a certain goal. The result of the transform subgoal was incorporated in the total sequence of operators that transformed the first to the final object.

The recursive powers of the list language in which GPS was written provided a natural way to realize the above scheme, in which each subgoal was kept in the total context of its supergoal and thus could be indefinitely particular. Each method was a routine that executed the problem-solving executive routine on the subgoals and then obtained its results as outputs:

```
Attain goal X:
    Select method (get M)
    Execute method M:
        Set up subgoal Y
        Attain goal Y
        Use product of Y to set up subgoal Z
        Attain goal Z
        etc.
```

It seemed so easy. We had finally provided ourselves with the freedom to use arbitrary goal types and arbitrary goal results; we had finally developed an organization for a general problem-solver. Actually, we had only mounted the other horn of the dilemma, for

the recursive structure dictated the order in which goals would be generated and attempted. The scheme implied by the structure described above is just the "depth-first" generation strategy of Fig. 1. The depth of generation was still controlled, for the routines actually looked like:

```
Execute method M:
    Set up subgoal Y
    Do we really want to try Y?
        If not, then quit method.
        If yes, go on:
    Attain goal Y
    etc.
```

Thus the goal tree could be pruned, so that unnecessary depth was avoided. However, there was only one chance to try goal Y: either it was tried when generated (thus growing the tree deeper) or the branch it represented was permanently abandoned.

We had lost the ability, available in the earlier programs, to throw all the goals into a pool and select arbitrarily which one should be attacked next. In LT, for example, the usual mode of operation was "breadth first." All the goals LT would ever generate at level 1 were obtained, then all those at level 2, then all at 3, and so on. In the Geometry program elaborate evaluation schemes were used to decide which of all the goals generated so far should be worked on next. Thus from an organizational viewpoint we had linked, in a complementary relationship, the freedom to select subgoals

with the variety of the goals. Increase in one implied decrease in the other.

But clearly we had not meant to give up the freedom of goal selection; we envisioned GPS growing its tree of subgoals in arbitrary fashion under the influence of content criteria that would select the best place to work. In the summer of 1959, having discovered that we had been seduced by the ease of writing recursive programs in IPL, we considered various schemes for resolving the dilemma. Here was a pure question of organization--to find a way that would give us both desirable properties, without having to pay too much for either. Only one of the schemes we considered has survived, but the others are worth recounting briefly.

SOLUTION BY REMOTE CONTROL

First we tried to maintain the recursive subroutine hierarchy. To work on a subgoal in this scheme, it was necessary to get into context on all of the higher subroutines. This requirement preserved the ability to deal with the results of arbitrary subgoals. To permit a goal to be tried (and retried) on various occasions under the control of the routines associated with higher goals, we violated the hierarchy temporarily to put control back into the hands of the higher routine. You are to imagine a method routine, having just created a

new subgoal, calling "Boss, I've got a new goal; should I try it?" and getting the answer "Yes, but call me if you run into more goals." Actually the scheme was much more automatic than this, for it did not involve an act of communication between routines, but rather the location in routine A of the control for what happened in the midst of subroutine B.

To illustrate this scheme, suppose GPS were executing method M on goal G and had just created a new subgoal, G'. Then M could execute the following instruction:

Attempt goal G'.

This constitutes a decentralized goal attempt, since M will not get control back again until the attempt is over. It exercises no control over how much effort is expended on G', what methods are used, etc. Alternatively, M could execute:

Attempt Goal G'; return control at first subgoal.

This constitutes a centralized goal attempt. Suppose a method, M', is applied to G' and generates a subgoal, G". When M' attempts it by means of one of the "attempt" instructions, control automatically returns to the point in M following the instruction above. Several options are now available to M. It could attempt G" by executing "Attempt goal G"." This action has no further reference to goal G' or its method M'. Alternatively, M could permit the action that it had interrupted to continue:

Attempt goal G'; return control at first subgoal;
Evaluate subgoal;
Attempt goal from prior context; return control
at first subgoal.

We have shown M as keeping control; it could have attempted the goal from the prior context--i.e., from M'--on a decentralized basis without asking for any return of control until the attempt of G" was finished. The inserted evaluation signifies that M may use arbitrary processes in deciding whether or not to attempt G". Suppose at this point G" were successfully attained. Then the results must be used by method M', which created G" and knows how to use it. Since control resides in M and not M', M would execute:

Execute prior context, to use results of goal.

Again, this is a decentralized action, since M did not set up the conditions under which control would revert to it.

The picture given above shows how central control was maintained while the goal tree was growing into new territory. It was also possible to re-attempt a goal at any time. No additional mechanism was necessary until a goal was attained, at which point GPS executed an instruction such as:

Execute original context, to use results of goal.

This instruction determined the method that had created the goal and re-executed it from the point following where the goal had been attempted originally.

In summary, this remote control scheme contained instructions for attempting goals, either directly or from their prior context--i.e., from the context that had just been interrupted. In addition it contained instructions for executing the routines that could use the results of goal attainment--the prior contexts or the original contexts. For all these types of instructions, there were centralized variants, which requested a return of control at the next subgoal, and decentralized variants, which let the subroutines go their way.

The great virtue of this scheme is in avoiding any restriction on how the decisions were to be made about goals. By passing control back to the higher routines we allow them to use arbitrary processes in order to decide what to do. Thus the control scheme does not seem to impose a substantive restriction. The difficulties are at least two. First, this organization suffers from the evil of large centralized organizations everywhere: although the top executives have the freedom to decide in any way they want, they don't have enough information about the local situation to decide wisely. They are hopelessly out of context. The second difficulty involves the piling up of administrative apparatus. As long as there is one controller, the system seems reasonable. But an indefinite cascade of controllers is possible, in which M requests the return of control from M', which

requests the return of control from M", which requests the return of control from ... and so on. Each decision to attempt a subgoal must pass through a long sequence of separate decisions, and the whole system begins to seem very cumbersome. Thus, the scheme was abandoned.

SOLUTION BY COMMUNICATION

An alternative to passing control back to the higher executive is to have the executive send messages down to the subordinates. This was the solution we tried next. This alternative still preserved the subroutine hierarchy, permitting arbitrary types of subgoals. The higher routines would formulate messages, and the lower routines would read them. The messages might instruct the lower routines to generate new goals, to retry old ones, to evaluate in such and such a way, to pass the message down to lower routines, and so on. The advantage of this scheme is that the acting routine can integrate the information in the message with the local context. The difficulties are again two. First, the messages must contain strategies, not just decisions. They must be in the form of partial information that can be combined with other information available to the local routine. We had no good formal language for this kind of communication. We developed a set of discrete questions, such as:

Are there any communications?
Do I control communications?

Do I attempt this subgoal?
Do I search for a subgoal to retry?
Do I continue, using the results of the goal just attained?

The executive working on a particular goal asked appropriate questions of the message received from the higher level, and obtained various answers upon which it based its actions. Although rather complex in its interpretation, this language was neither a flexible nor a rich vehicle to communicate strategies. The second difficulty is that GPS, operating in this fashion, resembles nothing so much as a bureaucracy bogged down in paper work--everyone busily writing messages that others have to read. Notice above that new organizational problems were created, such as who had the right to change the content of messages. So this scheme too was set aside.*

SOLUTION BY INTERPRETATION

Up to this point the appropriate metaphor has been a large human bureaucracy: each subroutine is an office; each has its own duties; control is decentralized among the offices; and communication is highly stylized and formal. Alternative metaphors are possible. The problem-solver should be a single personality, wandering over the goal net much as an explorer wanders over the country-

*Both the remote control scheme and the communication scheme were programmed as modifications of GPS as it then existed. Neither was debugged, since the defects had become sufficiently clear, once the details of the organization were worked out, to lead to their abandonment.

side, having a single context and taking it with him wherever he goes.

In this scheme the methods can no longer be routines. The active principle must lie in the central executive. Methods must be schema that the central executive can consult and follow, but where each important decision--to attempt a new goal, to go on with the next step of method--is made by the executive. This is the solution we have adopted in GPS-2.*

As shown in Fig. 3, each method is written as a sequence of segments. Each segment constitutes an action in which the executive has control and decides what to do--e.g., whether to execute the next segment. The executive is essentially an interpreter, the methods still being programs, but in a higher language. The GPS-2 executive is somewhat more sophisticated than a standard machine interpreter, which only has the functions of fetching the next instruction and executing it. Here, the interpreter receives information about the results of the segment's action and is able to make various decisions about what actions should be undertaken next.

Have we really attained both our objectives? Yes, to a degree. With the new scheme a goal may be

---

*GPS-1 was written in IPL-IV for JOHNNIAC. The changes implied in the interpretive solution were extensive enough to justify a complete recoding into GPS-2 in IPL-V. This latter organization is the one currently used.

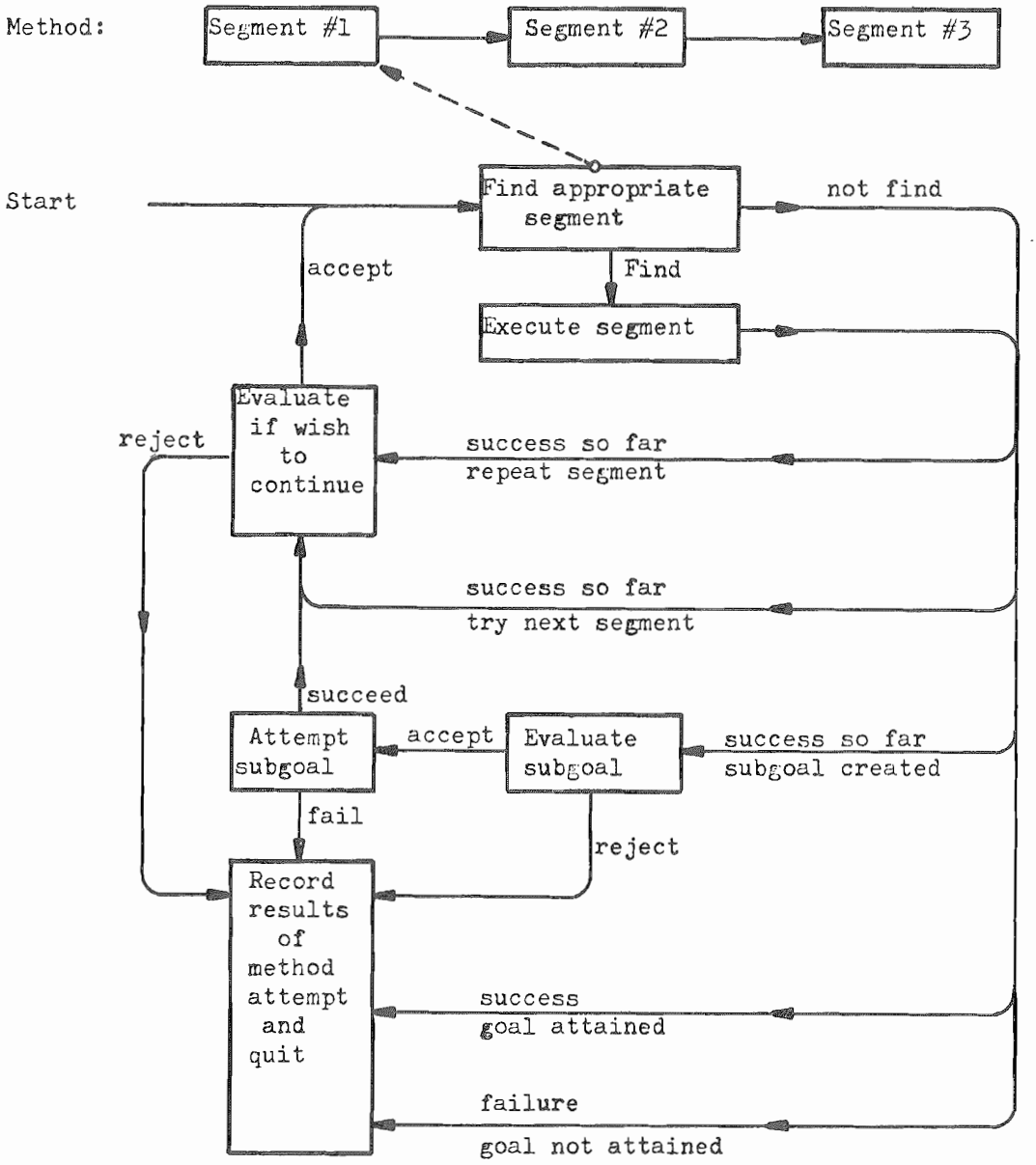


Fig. 3. Schematic flow diagram for executing methods in GPS-2.

selected anywhere in the goal tree and attempted; this can occur at any time and more than once. The methods still generate goals and contain the information about how to use them. On the one hand this means that new goal types and new uses of results can be introduced by increasing the number and variety of methods.* On the other hand it means that to use the results of a goal requires finding the segment of the method of its super-goal that is prepared to use it.

SOLUTION BY UNDERSTANDING

The information that the above interpreter obtains about the total situation is still impoverished. It still knows nothing of the nature of the methods--of what they really produce in the way of results and how they use them. It is still very like a blind man who has learned to push buttons to go his way, but only receives back a few taps to tell him where he is going. One would like a central process which could assimilate the knowledge of its environment and the techniques for manipulating it--that could understand how to use the results of its efforts. In attempting to create such a central organization we found--as we had with the problem of communicating

*All the methods coded to date have been representable as simple non-branching sequences of segments (with repetition of segments possible). The interpreter of Fig. 3 is specialized to this case.

strategies--that we had no concepts and no formal language to discuss the variety of results and their uses. The program shown in Fig. 3 can be viewed as a first attempt to expand the amount of knowledge that a problem-solving interpreter should have about its environment.

V. ACCESSING INHOMOGENEOUS COLLECTIONS OF DATA

List structures solved some of the problems of data representation for problem-solvers, but by no means all. For example, consider specifying a chess move. There is a man to be moved, designated, say, by the square he is currently occupying, i.e., the square from which the move takes place. This man moves to another square. Sometimes, but not always, this square has a man on it, who is then captured and removed from the board. Information on these points is enough to specify most moves, but several special cases exist. There are two castling moves, each of which requires moving both a King and a Rook. If the Pawn moves to the eighth rank, it is promoted and it is necessary to specify what piece it will become. Finally, there are en passant moves, in which a Pawn is moved one square diagonally--as in a capturing move--but the man is captured from a different square.

How shall we represent the information in a chess move? The question is peculiarly organizational. There is one imperative: we must get into the representation enough variety to discriminate all the different kinds of moves.* Yet there are many ways of satisfying this requirement. How shall we select one? Numerous routines

---

*Ashby even calls this the Law of Requisite Variety (25, pg. 206).

must use the representation: routines for making moves, for testing their legality, for generating them, for testing when two are the same, and so on. To some degree, each routine must be adapted to the representation. Depending on what representation is used, particular routines will be fast or slow, easy or hard to code, possible or impossible to change.

This problem of representation is not unique to problem-solving programs; it occurs in all programming to some degree. It becomes especially vexing when the information to be encoded consists of an inhomogeneous collection, as in the case of the chess move above. For orderly homogeneous information, such as vectors of numbers or sets of identical symbols, natural ways of encoding exist and few problems arise. Thus, the areas that are most vexed by the problem of representation are business data processing, information retrieval, and problem-solving, rather than numerical analysis.

Let us follow the chess move example a little further. A standard procedure for encoding arbitrary information is to create a set of "fields"--i.e., an interval of bits in a word (or words). Each field corresponds to some variable aspect of the move. The size of the field is set large enough to cover the range of possible conditions expected for the variable feature; these possibilities are somehow encoded into the bit patterns the field can hold.

This is possible for the chess move: assign one field for the From square, another for the To square, etc. A certain discomfort arises when space is set aside to encode the extra Rook moves for castling and the capture square for the en passant moves. These moves occur rarely, yet they may require the lion's share of space if one proceeds incautiously. In fact, if only a small part of the total possible information is present at any one time, assigning fixed fields for all of it is obviously the wrong solution. Most of the space will be empty.

This is a situation that list structures were meant to handle. In a list formulation, one can assign symbols to name the men, squares, etc. Then a possible representation would be:

Move List:

From square	
To square	
Special move symbol: castle, en passant, promotion.	
From square of Rook To square of Rook	for castle
Capture square	for en passant move
Man type promoted to	for promotion

Convention: if not a special move, list terminates after second list cell; if a special move, the remainder of the list is encoded according to the identifying symbol in the third list cell.

This has solved the space problem to some extent.

The routines that work with moves can use this representation only if they know it. That is, only if the routine knows that the first list cell holds the

From square; that the second holds the To square; that there are three special symbols, S1, S2, and S3, that can be in the third cell and that if it is S1, then the move is a castle, but if it is S2 then ... etc.--only if it knows all these facts can it perform its task. Knowing these facts, a routine has the necessary information to retrieve the symbols from the representation, interpret them, and perform its task.

Every routine must include both the processing necessary to generate its output and the processing necessary to deal with the input data representations. These are not separable, since the information must always be in some representation and the processes of the routine can only work on representations. Yet if we change the representation, something will change in the routine--but not everything. The routine and the representation also share the total information about the move, sometimes in a rather intimate way. In our example, the special symbols serve only to select one of three distinct subroutines associated with the three special cases. Each of these subroutines carries all the information about what to do with the extra information in its own case. Nothing in the data structure says what to do with any of the information. We could have put more information into the routine and less into the data; for example, we could have used a separate symbol to stand for each castle. Then the

Rook square information would have been unnecessary and the routine would have known just how to move the Rook (and the King too). We could even have used a separate arbitrary symbol for each possible move and simply assumed the routine could recognize each one individually.

The effects of this intimate dependence and division of information between routine and representation are well known. Modification and extension of a program is limited by how particular the routines have become. On a larger scale, a complex program with many kinds of data becomes a collection of special conventions and arrangements, each with its own routines and rigidities.

At least two directions have been pursued in trying to deal with this particularization of routines and data. One is to try to retain full particularity of the individual situation, but to attain some uniformity through data descriptions--i.e., through formats. As long as one deals in fixed formats, there can be a uniform language for writing routines, which is recoded at compiling time in terms of the specific details of the representation selected by the compiler. One gets the best of both worlds. This is the route being taken by work in business data processing. If the program is essentially dynamic, or if the data varies in amount as well as over the values of given variables, then data descriptions imply operating interpretively--i.e., through processes that consult

formats to determine how to interpret particular structures. (To my knowledge no one yet carried this latter effort very far.)

The second route has been taken in the work on problem solvers. From considerations quite like those we have been raising, the designers of list languages were led to provide an additional way of storing information besides storing on lists. In IPL-V, for example, the scheme can be stated as follows:*

> Let X, Y, and Z be any three symbols in the language. Then it is always possible to perform the following three processes:
>
> Establish the relation Y = X(Z)--i.e., associate Y with Z a/c X.
>
> Find X(Z) if it exists--i.e., the symbol associated with Z a/c X.
>
> Remove X(Z)--i.e., remove the association a/c X.

These processes form a complete system for reading and writing information: given some symbol, say Z, any other symbol, say Y, can be stored with it by means of an arbitrary association, X. Once stored, it can be read again by means of the "find" process or erased by means of the "remove" process.

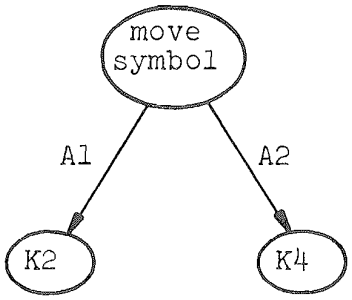
Given this scheme, a natural way of encoding a chess move would be to assign a set of symbols to stand for the

---

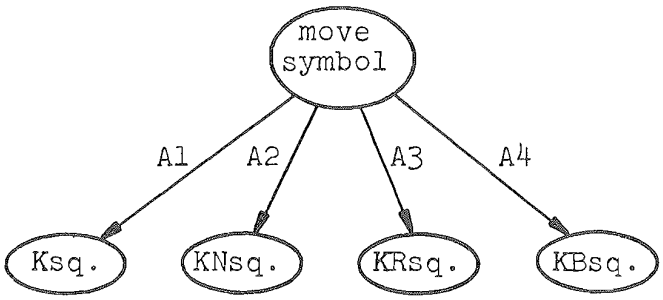
*These are the description list processes of IPL-V, J10, J11, and J14. Details about their realization in terms of list structures can be found in the IPL-V Manual.(17)

various subpieces of information: A1 for the From square, A2 for the To square, A3 for the secondary From square, A4 for the secondary To square, A5 for the new-man-type, and A6 for the capture square for en passant. These would be the attributes--the symbols according to which association would be made. A move would be a symbol that had the prescribed associations. Figure 4 shows how a standard move and a castling move would be described in this representation. Only those associations are stored that are needed, just as in the list representation. The special nature of the move, if any, is detected by the existence of the special associations.

This association scheme provides a uniform way of encoding inhomogeneous collections of information. Each type of information is simply assigned a name and called for by that name. This call is relative to the basic symbol (the move), just as in table look-up procedures the index is relative. In fact, the scheme is just a table look-up that is unrestricted with respect to what is index, what is entry, and when the decision has to be made that the table shall hold an entry. All the components of an association are symbols, and to that degree homogeneous and restricted; but since in a list language symbols can name arbitrary structures, this restriction is in fact no restriction. This uniformity is paid for by searching for each item when it is needed. Since the symbols the routine



Opening move with King's Pawn from square K2 to square K4 (P-K4).



Castling to the King's side: King moves from K square to KN square; Rook moves from KR square to KB square.

Fig. 4. Use of Associations to Encode Chess Moves.

knows--the names of the information it is interested in--bear no structural relation to the way the information is stored (and they cannot, since the symbols can be selected arbitrarily) there must be a process that discovers the relationship--e.g., through searching the list that serves as a table.

One may question whether such uniformity deals effectively with the organizational problems of inhomogeneous data and the multiplicity and particularity of encodings that arise in large systems. There is some empirical evidence that it does. This association scheme was introduced into IPL as an augmentation to lists: it was thought that besides the list structure, there might be some additional "descriptive" information. Hence, in the large programs that have been coded in IPL both representational schemes have been freely available--in free competition, so to speak. There has been a continual increase in the use of the association process to encode everything but homogeneous structures: the programming is simpler, encoding decisions are avoided, etc. The extreme example is the current version of GPS, whose data is represented almost entirely by associative structures. To give an illustration of the effect of this decision, goals and logic expressions, the two main kinds of structures in GPS, are now handled uniformly. The same match processes are used to match two goals and to match two logic expressions.

How far this associative memory structure goes towards handling the problems raised in this example is unclear. Data must always be represented, and knowledge must exist about this representation in the processes, thus making them to some degree representation-dependent and not easily modified. Even with the associative processes, the routines still seem quite particular. There is a long way to go to achieve full generality.

VI. PRESERVING PAST HISTORY

GPS has goals; they are rather elaborate data structures of the associational variety discussed in the last example, containing a wealth of heterogeneous information. Each goal represents a single state of desire; GPS is under the control of a single goal at any instant, working on it, preparing to quit and go back to a higher goal, or about to create a new subgoal and go deeper.

Why should GPS have goals? They are expensive to produce and use up quantities of space. They are probably the single biggest reason why GPS appears to be a plodder--all its time seems to be spent in building goals. When GPS goes into the context of a goal, it uses a set of working cells (about a hundred). It pulls information off the goal and puts it into these cells for more immediate reference. As new information is generated it is put into these temporary cells as well as being stored away on the goal structure. Why maintain a goal structure? Why not use just the immediate storage--i.e., only the currently needed information?

The most general answer, perhaps, is the need to remember the past. But there are several quite different reasons why it must be remembered. Perhaps the most cogent, given the discussion so far, is keeping information about the supergoals still in progress. This is

the same as the reason why there must be a push-down list for recursive routines: as each subpart is completed the higher routine must be reactivated at the right point. But as we saw earlier, if this was all there was to it, we could use the recursive capabilities of list languages much as in any subroutine hierarchy, without creating separate goal structures.

A second reason for retaining goals is to remember the terrain so as not to go over it again. Under very special strategies or with special tasks the generation of subtasks will never repeat itself. In general, however, checking is required to avoid repeating. Sometimes this is only a heuristic matter--a question of whether the search will go faster or slower. More often it is a crucial matter of avoiding a cycle.

The third reason for remembering, and the focus of this example, is to return to a point reached in the past in order to try something different from that point. From this viewpoint, a goal is a place marker, noting the possibility that from its location something of significance can be done. Turning the matter around, if the problem-solver has arrived at a choice-point--where one thing must be tried and others put aside--then preserving the opportunity to later make the other choices if the first one fails implies the recollection of this

situation. If implies that a goal must be created at this point.* This seems a simple matter. But as stated earlier, setting up goals and filing information away on them is a major housekeeping activity for GPS. Consequently, we have tried several dodges to get around the implications of the proposition.

One of these is the concept of immediate operators. Suppose, as happens frequently in applying operators, we compare the expression A with P.Q, where P and Q are constants and A is a variable. There is a difference--the expressions are manifestly not the same--but a difference we know precisely how to eliminate: substitute P.Q for A. According to GPS liturgy, when a difference is found a subgoal is created to reduce the difference. This subgoal results in the selection of an operator (in our case the substitution operator) and a goal is set up to apply this operator. If it is successful (as we know it will be), then the result of applying the operator (P.Q) is used as the result of the reduce difference goal. This leads to setting up a new subgoal to see if the modified expression (P.Q) is the same as the criterion expression (P.Q) (as we know it will be). Finally we are permitted to say that the two are the same.

---

*Nothing is implied about whether the goals must be homogeneous--i.e., all contain the same information or have a common format. This issue is related to some of the other examples.

It seems a lot of unnecessary work to go through all this. Instead, whenever GPS finds a difference it asks if there is any "immediate" operator that applies to it. If it finds one--as it does for substituting an expression for a variable--then it applies this operator forthwith and proceeds. Three subgoals and a large amount of processing have been avoided.

But there are difficulties. On one occasion GPS matched $A \supset -R$ against $R \supset Q$. It promptly substituted R for A, thus getting $R \supset -R$ against $R \supset Q$, with which it could do nothing. Of course, it should have first detected the difference in position, so that it could have changed $R \supset Q$ to $-Q \supset -R$. By charging ahead and making the substitution, it foreclosed its chance of taking that action--even of taking it after it had detected the mistake in the substitution. (If GPS had delayed, it would have seen the right difference.)

To illustrate the same issue in a different guise, consider the problem of applying operators that have more than one input. These arise in logic, for example, in the transitive law: from $A \supset B$ and $B \supset C$ produce $A \supset C$. Each of the two inputs is an independent expression. Normally in GPS only one expression is at hand to be input at the moment when the law is to be applied. Typically there is a goal such as: apply $(A \supset B, B \supset C \Rightarrow A \supset C)$ to $P \supset Q$. It is necessary to find a second input

expression and to decide in which order to assign them to the two input forms. The goal tree developed by GPS in doing this is shown in Fig. 5. The top goal is to apply the two-input operator to a single expression. This leads to a difference, since a set of objects is being compared with a single object, and a subgoal is created to reduce this difference. This leads, by a short cut similar to the immediate operators, to direct application of a selection routine, which selects the member of the set that is most similar to $P \supset Q$. In the example either member will do, and the first is chosen. This leads to applying the initial operator again, but only fitting the first input form to $P \supset Q$. This is easily done with two substitutions. However, the result is not a new object, but a new specialized operator, of the form $Q \supset C \Rightarrow P \supset C$. This operator is not valid in general, but only within context where $P \supset Q$ is a true expression. At this point, GPS needs to find the second input to the operator. It has available a set of admissible expressions, so it creates the goal of applying the operator to the set. This again leads to a difference, which leads to the selection of the expression most similar to $Q \supset C$. Notice that the prior incorporation of $P \supset Q$ gives enough additional information to select $Q \supset R$ rather than $R \supset S$. Finally the special operator is applied to $Q \supset R$ and a final result, $P \supset R$ is obtained.

1. Apply $(A \supset B, B \supset C \Rightarrow A \supset C)$ to $P \supset Q$

    2. Reduce Set-versus-object difference between $(A \supset B, B \supset C)$ and $P \supset Q$

        Select: $A \supset B$

    3. Apply $(\underline{A \supset B}, B \supset C \Rightarrow A \supset C)$ to $P \supset Q$

        Produce specialized operator:
        $P \supset Q, \underline{Q \supset C} \Rightarrow P \supset C$

    4. Apply $(P \supset Q, \underline{Q \supset C} \Rightarrow P \supset C)$ to $(R \supset S, -Q.P, Q \supset R)$

        5. Reduce Object-versus-set difference between $Q \supset C$ and $(R \supset S, -Q.P, Q \supset R)$

            Select: $Q \supset R$

        6. Apply $(P \supset Q, \underline{Q \supset C} \Rightarrow P \supset C)$ to $Q \supset R$

            Produce final object: $P \supset R$

Fig. 5. Application of Two Input Operator by GPS.

Again, it seems a lot of work. But each of these goals represents an opportunity for error. In earlier attempts to use multiple input operators in GPS we short circuited goals 2 and 5, making the selections directly. When the direct selections were wrong the chance of coming back and making different selections had been foreclosed. With respect to the other goals, if $-P \vee R$ had been given instead of $P \supset R$, or $-Q \vee R$ instead of $Q \supset R$, additional subgoals would have been required under goals 3 and 6 to change the connective.

So far only one side of the coin has been presented: rich goal nets are needed to preserve the opportunity for choice. Without this the problem-solver will be rigid, unable to try things and profit from its failure. But there are difficulties. A price is being paid in cumbersome structures. And numerous anecdotes warn of the dangers in a glut of information. LT provides a nice example.(6) It first found a proof of a certain theorem given all the prior theorems in the chapter. However, when it was given only the axioms plus one other theorem and presented with the same problem, it found a longer proof in about a third of the time. The additional theorems required more processing than they were worth.

Perhaps a second example will reinforce the point. Recently we programmed GPS to try the well known puzzle

of the Missionaries and the Cannibals.* We started from the program for logic, on which we had been working. In the latter program all the intermediate products are kept, both for checking and to provide inputs to the two-input operations. GPS solved the problem. Then, we noted that our human subjects, working with a physical model in front of them, seemed only to recall the initial position and the current one. We simulated this situation, giving GPS an external representation and having it adopt a strategy that didn't remember any of the intermediate positions. GPS solved the problem this way much faster--all the extra intermediate goals and expressions had given it only useless things to worry about. It was better off forgetting them completely.

To draw the moral, a fixed regime of goal building is to be avoided as are all other rigidities. Only those goals should be built that seem absolutely necessary; the rest provide nothing but noise and extra processing. To do this requires developing strategies for what of the past should be remembered. But a prior requirement is the capability to stipulate arbitrarily whether or not a goal should exist, and when a goal should be destroyed. It

---

*Three missionaries and three cannibals are on one side of a river. They wish to cross to the other side but have only a single boat that holds two people. The problem is to get everyone across without letting the cannibals ever outnumber the missionaries on either river bank.

must be possible to forge ahead, heedless of the memory problems, until it becomes clear that some difficulties exist, then to back down to the previous goal (now serving as an anchor point) and to proceed again more cautiously, creating suitable intermediate goals.

This capability is a matter of organization. It is a question of discovering the role played by various fixed conventions in the current GPS--of trying to shift more information from the goal routines into the goal structure so that less need be assumed in the routine as fixed and immutable. This problem is hardly insurmountable; even to put the question is to half solve it. GPS will eventually have much greater freedom with its goal building. How far the solution will go towards the general problem indicated here is harder to predict.

VII. CONCLUSION

We have now seen a handful of examples of organizational problems connected with building problem-solving programs. The issues represented seem diverse, the solutions particular. Some seem to be simply the projection into the domain of problem-solvers of issues that plague all programming. Others seem special to building intelligent machines. Can anything be said about them generally?

First, let me repeat that the primary purpose of this paper is to set out some examples of organizational problems, describing each in its own terms. Although there exists no adequate framework for describing them, hopefully enough of a picture has been given to permit their recognition in other complex programs if they exist there. Such problems are seldom recorded in print, presumably because we neither know how to talk about them nor how to assess their significance. The faith of the botanizer is that specimens precede classification, which in turn precedes theory.

Second and more tentatively, we can discern some pattern in these examples. As we observed earlier, a program can operate only in terms of what it knows. This knowledge can come from only two sources. It can come from assumption--from the programmer's stipulation that such and such will be the case. Alternatively, it can

come from executing processes that assure that the particular case is such and such--either by direct modification of the data structures or by testing. Now the latter source--executing processes--takes time and space; it is expensive. The former source costs nothing: assumed information does not have to be stored or generated. Therefore the temptation in creating efficient programs is always to minimize the amount of generated information, and hence to maximize the amount of stipulated information. It is the latter that underlies most of the rigidities. Something has been assumed fixed in order to get on with the programming, and the concealed limitation finally shows itself. In LT the convention that "the solution of a subgoal must solve the main goal" is an example.

When the rigidity becomes too gross to neglect, there are two paths towards resolution. One is to relax the constancy of the convention and allow a fixed number of cases. The program is expanded to contain separate subparts for each case and testing programs are added to transfer control to the correct part. The normal fracturing of a program into many parts depending on the type of data is the result of this sort of solution. The virtue of this technique is that it can always be done at least to some extent. No new synthesis is required, only the willingness to write a more complex program with more parts. And this is its chief vice: it makes programs

ever more particular, ever more fractured, ever more resistant to further correction and modification, ever more obscure and complex.

The other path to solution is again to remove the rigidity--to let be variable what once was fixed--but to discover a new set of invariant concepts to deal with this variability, so that the complexity of the program does not increase. The list processes--insert, delete, etc.--represent an excellent example of this. The use of the associative processes to handle inhomogeneous data collections is another. The attempts at a set of processes for remote control, at communicating strategies, and at a description of understanding how to use results, are abortive or incomplete attempts.

Progress is a see-saw between these two types of solutions. The uniform goals in GPS are partly an attempt to avoid separate structures for all the particular kinds of information that need to exist. Strong uniformities were accepted in the goal scheme to make it operational. Some of the limitations of the current scheme are now quite clear, and are forcing a solution with more variability again. Along the way common routines for handling all the goals have been achieved, an intermediate success not likely to be lost.

It seems also that as we diminish the amount of assumed information, and avoid having separate parts of

the total program know separate things, we are moving toward more interpretive schemes. It is one way to get dynamic variability without increasing complexity too much. One good example is the current GPS, as indicated in Fig. 3. Additional examples could be given from the other parts of GPS, where the matching process is now accomplished by a higher level interpreter, as is the similarity testing that makes the selections shown in Fig. 5.

All this seems quite reasonable. Roughly, we are trying to shift the information from the programmer's head, which is where programming efficiency for simple algorithms says it should be, to the dynamic data structure. Surely this is right. For if ever a fully capable intelligent program is realized, it will be recognized by noting that it can get along without any programmer at all. Then all its information will be in the program structure and none in the programmer's head.

REFERENCES

1. Samuel, A. L., "Some Studies in Machine Learning Using the Game of Checkers," IBM J. Res. and Develop., Vol. 3, No. 33, July 1959, pp. 210-229.

2. Strachy, C. S., "Logical or Non-Mathematical Programmes," Proceedings Assoc. for Computing Machinery, September 1952, pp. 46-48.

3. Bernstein, A., et al., "A Chess-Playing Program for the IBM 704 Computer," Proceedings of the 1958 Western Joint Computer Conference, March 1959, pp. 157-159.

4. Kister, J., et al., "Experiments in Chess," J. Assoc. for Computing Machinery, Vol. 4, No. 2, April 1957, pp. 174-177.

5. Newell, A., J. C. Shaw, and H. A. Simon, "Chess-Playing Programs and the Problem of Complexity," IBM J. Res. and Develop., Vol. 2, No. 4, October 1958, pp. 320-335.

6. Newell, A., J. C. Shaw, and H. A. Simon, "Empirical Explorations of the Logic Theory Machine: A Case Study in Heuristics," Proceedings of the 1957 Western Joint Computer Conference, February 1957, pp. 218-230.

7. Gelernter, H., J. R. Hansen, and D. W. Loveland, "Empirical Explorations of the Geometry Theorem Machine," Proceedings of the 1960 Western Joint Computer Conference, May 1960, pp. 149-150.

8. Tonge, F. M., A Heuristic Program for Assembly Line Balancing, Prentice-Hall, 1961.

9. Slagle, J. R., "A Heuristic Program that Solves Symbolic Integration Problem in Freshman Calculus (SAINT)," Unpublished Ph.D. dissertation, MIT, June 1961.

10. Newell, A., J. C. Shaw, and H. A. Simon, "Report on a General Problem-Solving Program," Information Processing: Proceedings of the International Conference on Information Processing, UNESCO, June 1959, UNESCO Paris, 1960, pp. 256-264.

11. Minsky, M., "Steps Toward Artificial Intelligence," Proceedings of the IRE, Vol. 49, No. 1, January 1961.

12. Carr, J. W., III, "Recursive Subscripting Compilers and List-Type Memories," Communications of the ACM, Vol. 2, No. 2, February 1959, pp. 4-6.

13. Gelernter, H., J. R. Hansen, and C. L. Gerberich, "A FORTRAN-Compiled List-Processing Language," J. Assoc. for Computing Machinery, Vol. 7, No. 2, April 1960, pp. 205-211.

14. Green, B. F., "Computer Languages for Symbol Manipulation," IRE Transactions on Human Factors in Electronics, HFE-2, No. 1, March 1961, pp. 25-33.

15. McCarthy, J., "Recursive Functions of Symbolic Expressions and their Computations by Machine," Communications of the ACM, Vol. 3, No. 4, April 1960, pp. 184-195.

16. Newell, A., "Notes for Lectures on Heuristic Programs," Applications of Logic to Advanced Digital Computer Programming, Intensive Summer Sessions, University of Michigan College of Engineering, August 1957.

17. Newell, A., (ed.), Information Processing Language-V Manual, Prentice-Hall, 1961.

18. Newell, A., and J. C. Shaw, "Programming the Logic Theory Machine," Proceedings of the 1957 Western Joint Computer Conference, February 1957, pp. 230-240.

19. Perlis, A. J., and C. Thornton, "Symbol Manipulation by Threaded Lists," Communications of the ACM, Vol. 3, No. 4, April 1960, pp. 195-204.

20. Shaw, J. C., A. Newell, H. A. Simon, and T. O. Ellis, "A Command Structure for Complex Information Processing," Proceedings of the 1958 Western Joint Computer Conference, May 1958, pp. 119-128.

21. Weizenbaum, J., "Knotted List Structures," Communications of the ACM, Vol. 5, No. 3, March 1962.

22. Selfridge, O. G., "Pandemonium: A Paradigm for Learning," Proceedings of Symposium on the Mechanization of Thought Processes, HMSO, London 1959, pp. 511-529.

23. de Groot, A. D., Het Denken van Den Schaker, Amsterdam, 1946.

24. Green, B. F., et al., "Baseball: An Automatic Question Answerer," Proceedings of the 1961 Western Joint Computer Conference, May 1961, pp. 219-224.

25. Ashby, W. R., An Introduction to Cybernetics, Wiley, 1956.