MASSACHUSETTS INSTITUTE OF TECHNOLOGY
A. I. LABORATORY

Artificial Intelligence
Memo No. 255A

April 1972

WHY CONNIVING IS BETTER THAN PLANNING

Gerald Jay Sussman

and

Drew Vincent McDermott

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-70-A-0362-0003.

Abstract

This paper is a critique of a computer programming language, Carl Hewitt's PLANNER, a formalism designed especially to cope with the problems that Artificial Intelligence encounters. It is our contention that the backtrack control structure that is the backbone of Planner is more of a hindrance in the solution of problems than a help. In particular, automatic backtracking encourages inefficient algorithms, conceals what is happening from the user, and misleads him with primitives having powerful names whose power is only superficial. An alternative, a programming language called CONNIVER which avoids these problems, is presented from the point of view of this critique.

Acknowledgement:

We must deeply acknowledge the profound influence of Joel Moses on this paper. Some of the ideas here are directly due to him; others were independently arrived at by him. Most of our ideas were arrived at by observation of real users of MICRO-PLANNER. We thank especially Carl Hewitt, many of whose structures we used in the design of CONNIVER. Dan Bobrow, Jeff Rulifson, Bob Balzer, Chris Reeve, Marvin Minsky, Seymour Papert, Tom Knight, Terry Winograd, Richard Greenblatt, Donald Eastlake, David McDonald, Jon L. White, and William Gosper provided valuable sounding boards for these ideas.

The Problem with PLANNER

A higher level language derives its great power from the fact that it tends to impose structure on the problem solving behavior of the user. Besides providing a library of useful subroutines with a uniform calling sequence, the author of a higher level language imposes his theory of problem solving on the user. By choosing what primitive data structures, control structures, and operators he presents, he makes the implementation of some algorithms more difficult than others, thus discouraging some techniques and encouraging others. So, to be "good", a higher level language must not only simplify the job of programming, by providing features which package programming structures commonly found in the domain for which the language was designed, it must also do its best to discourage the use of structures which lead to "bad" algorithms.

PLANNER[1] is the language designed by Carl Hewitt of the MIT Artificial Intelligence Laboratory. (A subset of PLANNER was rather haphazardly implemented by G.J. Sussman, T. Winograd and E. Charniak. We call this operational system MICRO-PLANNER[2].) PLANNER incorporates three basic ideas; automatic backtrack control structure, pattern-directed data-base search, and pattern-directed invocation of procedures. Basically, backtracking is a way of making tentative decisions which can be taken back if they don't pan out. The pattern-directed data base search allows the user to ask for the data items called <u>assertions</u> which match a given pattern, and is intimately linked via the GOAL function to

pattern-directed procedure invocation, which gives the user the ability to say "Find and run a program whose declared purpose matches this pattern." This type of program, called a theorem, is expected to instantiate the pattern (succeed), and thus simulate an assertion. In fact, it simulates a whole class of them, since failures back into any such theorem cause it to make different choices and succeed with different instances.

How these mechanisms are related can best be illustrated by an example. The statement (GOAL (?A IN ?B)) is expected to assign the question-marked variables that do not have values already, or fail if it can't, causing a backup to the last decision point in the program.

GOAL instantiates its pattern by matching it against successive assertions, like (BLOCK2 IN BOX1). If it finds none, or enough failures propagate back to the GOAL to use up those it has found, it call theorems with matching patterns, such as:

```
(CONSEQUENT (X Y Z) (?X IN ?Y)
        (GOAL (?X IN ?Z))
        (GOAL (?Z IN ?Y))   )
```

which expresses one facet of the notion that IN is transitive. A PLANNER program executing (GOAL (BLOCK2 IN ?B)) first checks to see if it "knows" the answer, and if so sets B to it. If not, it binds X to BLOCK2, links Y and B, enters the theorem, and looks for a Z containing BLOCK2 and contained in some Y. Its net effect is to assign a value to B.

If a failure propagates back into the theorem, it finds another Y containing Z, and hence generates another ß; enough failures to use up those Y's drive it to find another Z;and a few more will make it and the original GOAL fail themselves. Backtrack control structure is the heart of this apparatus.

Automatic backtracking is implemented as follows: A PLANNER program, as it runs, grows a chronological stack of failpoints each of which corresponds to either a side effect or a decision point (a place where a choice is made between several alternative possibilities). A failpoint carries with it all information necessary to reconstruct the state of the running process at the time the failpoint was made. It may logically be considered to be a snapshot of that process (though it really saves much less than a copy). At some time, the process may decide to fail, perhaps because some decision made at a previous decision point led the program into a blocked state from which there are no viable alternatives. The failure then pops off the latest failpoint on the chronological stack. If this failpoint was a side effect, then it is undone, and the process continues failing. If this failpoint was a decision point, then if there are any remaining alternatives, execution proceeds from that failpoint with the next choice taken, or if there are none the failure continues to propagate. In these terms, GOAL finds exactly one assertion or theorem each time it is reached, but sets a failpoint to regain control if a failure should occur later.

For some time we have been studying PLANNER and the uses to which it has been put, hoping to learn just what modifications would be desirable to the user community. These investigations have led us to decide that this basic control structure of PLANNER is wrong, though its successes indicate that it contains many powerful (and seductive) ideas. This investigation has led to the design and implementation of a new, and hopefully cleaner language, CONNIVER[3], built partially on the good ideas found in PLANNER.

Here is our thesis: automatic backtracking, which occupies a place in PLANNER analogous to to that of recursion in LISP, is the wrong structure for the domain for which PLANNER was intended, that is, Artificial Intelligence. We argue that:

1. Programs which use automatic backtracking are often the worst algorithms for solving a problem.

2. The most common use of backtracking can almost always be replaced by a purely recursive structure which is not only more efficient but also clearer, both semantically and syntactically.

3. Superficial analysis of problems and poor programming practice are encouraged by the ubiquity of automatic backtracking and by the illusion of power that a function like GOAL gives the user merely by brute force use of invisible failure-driven loops.

4. The attempt to fix these defects by the introduction of

"failure messages" (to be explained) is unnatural and ineffective.

Thus we contend that the problem with PLANNER is automatic backtrack control structure. We must stress, however, that PLANNER has introduced many valuable constructs into our way of thinking, the most important of which is pattern-directed search of a hierarchical data base.

Note also that we are not contending that good programs cannot be implemented in PLANNER; that would be absurd. We are only claiming that PLANNER gets in the user's way when he tries to embody certain straightforward concepts in his programs. Nor are we making the weak point that PLANNER occasionally lures foolish programmers into inefficiency. One could try to make this criticism of LISP|4 by pointing out, for example, how it tempts one to write an exponentially exploding, doubly recursive algorithm for computing the nth element in the Fibonacci sequence:

```
(DEFUN FIB (N)
   (COND ((= 0 N) 1)
         ((= 1 N) 1)
         (T (+ (FIB (- N 1))
               (FIB (- N 2))))   ))
```

The language has led us astray here, since it discourages writing the iterative algorithm, but this is no condemnation of LISP; the mechanism of recursive control structure, although the wrong one to use in this pathological case, is often both the most natural and the most efficient

control structure for the problems of symbolic manipulation that are typical of LISP applications. PLANNER, however, almost forces inefficiency in exactly the applications it was designed for.

We now consider our points in detail.

1. All will readily admit that a perfectly clever program would do no backtracking; it would know where it was going at each step and never need to undo a bad decision. Good programs that know the structure of the problem domain (such as Moses' SIN|5|) have no need for an ability to thrash about, searching for a good approach (as in SAINT|6|). Pure backtracking (without failure messages) is essentially a mechanism for easily undoing a bad decision in the hope that a better alternative will be found. Thus it is most appropriate to algorithms which make such bad decisions either because of lack of sufficient guiding structure in the problem space or of sufficient knowledge of that structure in the program.

It is, of course, impossible in practice or in principle to achieve perfect or even adequate knowledge in most AI application programs. Inevitably, programs will recognize that they have gone seriously agley, and will have to undo part of what they have done. Unfortunately, pure backtracking undoes everything since the last decision, without enquiring as to whether it was the one at fault. Such a program will eventually stumble upon the right path, but its

organization makes it hard for it to learn something from an attempt that failed and erased all its side effects. The only attempt at correcting this intrinsic defect of failure in the PLANNER sense is the failure message device, to be discussed under point four.

2. Observation of the MIT vision group's[7] use of MICRO-PLANNER tends to indicate that one of the more important uses of backtracking, in programs which are not searching because they know exactly where they are going, is in information retrieval. Although important, it is curiously quite trivial for such a powerful mechanism as the GOAL primitive. Vision programs maintain large data bases of information about a visual scene, and often must be able to search out relevant data items from a mass of irrelevancies. For example:

```
(GOAL (?X IS BIG))
(GOAL (?X IS GREEN))
(GOAL (?X ON ?Y))
(GOAL (?Y IS BLUE))
(stuff ?X ?Y)
```

means "do the stuff" on the first objects X and Y such that "the big green X is on the blue Y." If stuff doesn't like the first ones found, it can easily fail to get more, if there are any. Note that what is going on here is sequential filtering of the possible assignments of X and Y by pattern-directed search of the data base and theorems. We see that backtracking must be used here because any particular big X chosen on line one may not be green, or may not be on something blue. The stack frame of each goal statement thus maintains a list of the hitherto

untried possibilities and if a failure reaches it, it tries the next one and proceeds.

A much more straightforward and revealing approach would be to use ordinary recursive and iterative control structure to filter the possibilities directly. Thus, for example, a LISP function FOR-ALL might be written, such that:

```
(FOR-ALL (?X IS BIG)
     (FOR-ALL (?X IS GREEN)
          (FOR-ALL (?X ON ?Y)
               (FOR-ALL (?Y IS BLUE)
                    (stuff ?X ?Y)))))
```

would have the desired effect. Here, FOR-ALL is just a standard LISP function which, upon entry, looks up all of the assertions and theorems matching the pattern given as its first argument (with values substituted for variables which are already assigned). It then assumes the first possibility, assigning variables appropriately, and evaluates its second argument. If the evaluation ever returns, rather than exiting the loop, the first element is removed from the list of possibilities and the process repeats. Notice that by appropriately nesting our loops no backtracking is required in the data retrieval. Here stuff is done on each X and Y which satisfies the criteria until stuff decides it has had enough, and leaves the nest of FOR-ALL's (with a RETURN, GO, or something similar).

This good nesting of loops has decided advantages. Besides being more efficient than backtracking (a marginal advantage), good nesting makes the scope of the action clear. There is no chance an unexpected

failure will propagate back into this code and compute without our explicit programming of this activity.

We want to emphasize that this insidious problem is not made up. It is observed by real users of MICRO-PLANNER who complain that they just can't control their programs because what they do depends on events before and after they are called. Usually any choice made in a piece of code doing such filtering eventually fails for the same reason that the first choice did, but backtracking tends to treat all decision points as equally important and tries all possibilities; the only subsequent symptom that the program is running amok is that it takes forever to tell you it failed. The consequent theorem given suffers from exactly this problem; if called by, e.g., (GOAL (BLOCK2 IN BOX1)), its only possible actions are either to find a Z between BLOCK2 and BOX1, or to fail. Although which Z is found cannot possibly affect subsequent events, a failure back to the theorem will cause it to look up another Z, succeed, and allow its caller to fail again in exactly the same way!

One way out of this problem is to FINALIZE the program from just before the first filter to just after the code which the user doesn't want reentered upon failure, thus freezing all its side effects once and for all. This is unsatisfactory because often some of them should be undone upon some later failure.

In some cases this is a problem, in some cases not; what is always a problem is that the structure of a PLANNER program does not

reveal what the programmer's intentions are. He must always keep in mind that in effect there is only one gigantic nest of failure-driven loops in any PLANNER program, and every subprogram that might fail is only a tiny piece of it. We think that it is essentially clearer for any looping or nesting structure to be made explicit.

3. As PLANNER is currently organized, it provides a very compact notation in which to encode exhaustive searches for solutions to problems the programmer understands poorly. Other program organizations, though certainly possible, are clearly more complex and less transparently described. To overcome this difficulty, a multiprocessing capability has been patched into PLANNER in later versions, but several backtracking processes do not seem any better than one. Multiprocessing allows "breadth-wise search" of a sort, but it is only an abortive step to freedom by the poor programmer. Each of his processes is still crippled by the exhaustive searches built into system primitives; he must still spend time calculating the possible directions from and circumstances under which control could enter each line of his code. With all the machinery hung on his programs to circumvent the control structure, they look much less understandable; most programmers just can't be troubled. There is really no reason why they should be. We have already made the point, which we can carry much further, that a more revealing control structure would allow programmers to express a broader range of concepts more naturally.

There is a deeper issue here than what is needed in PLANNER to make it more powerful; we ask instead what it is about PLANNER that makes it so unusable. Its defaults are chosen throughout so that backtracking must be tediously reckoned with in every case unless the user explicitly prevents it. It is easy to say (as some PLANNER advocates do) that people should write their programs to avoid the temptation to backtrack except when necessary, but it is much harder actually to do it when the language gives them every opportunity to fail.

4. In order to give the user a modicum of control over the backtrack mechanism, failure messages were incorporated into PLANNER early in its history. The intent was to give a program the ability to fail with any message to a specific point which it has set up beforehand to catch the failure by matching that message. This does not give the user the ability to perform even the simplest of control functions. Suppose, for example, we have a goal which invokes a theorem. This theorem, in probing the search space, discovers something relevant to its further exploration. It would like to edit the list of theorems which are pending in the goal which called it (the alternatives which will be tried if the current theorem fails), deleting some entries and inserting others. It might even wish to sort the list of alternatives according to some general criterion. It has not yet, however, failed, and thus cannot return a failure message. Furthermore, it cannot get at the list of alternatives pending on its failure.

This is not a fine point of control structure theory; it would be extremely relevant to a PLANNER encoding of a chess program like that of Greenblatt[8]. For example, this program, in an analysis of a move, may discover that it is in danger of being forked. This discovery must change the whole set of criteria by which it judges further alternatives. It must try to make a move which meets the discovered threat, if possible.

We have been concentrating here on the sloppy interface between failure messages and GOAL, but there is a fundamental difficulty with them that would be encountered even if the user abandoned GOAL altogether. That is, they can't carry enough information. There is no way to fail back with the message: "Process P ran into difficulty T," because process P and its context have been destroyed by the failure. So all the relevant information must be contained in the T part of the message: "Difficulty T." It is clear that including all and only the relevant information is as impossible a job for a subroutine as anticipating the form of every possible failure is for its caller. In fact, the THMESSAGE primitive of MICRO-PLANNER has never been successfully used; it seems to be one of those superficially good ideas that prove to be unworkable.

It seems that a failing program has no choice but to make too much information frozen in too global a context, or to flush everything it has discovered and bet all its chips on one message it hopes somebody,

somewhere, can figure out. These alternatives do not really alter the blind nature of a failure-driven process, or of several of them. This is probably why they go unused.

At this point it is desirable to abstract our entire discussion away from the particular primitives of PLANNER, and enquire what is gained and what is lost by the use of automatic backtracking. What is gained is clear, and very appealing. In the first place, it provides a mechanism for generating alternatives, one at a time, to be used in an effort to accomplish some task. Secondly, it provides a mechanism for eradicating the consequences of accepting an alternative later found to be unsuitable.

We have criticized the consequences of this scheme in several ways already. Now we shall argue that its basic defect is that it forces the dangerous assumption that the alternatives at each decision point are independent; that (as within all PLANNER primitives) the trial of one of them may produce little or no information which can influence the selection of further alternatives, or the way in which they are run. This is enforced by the eradication of the consequences of a hypothesis when that hypothesis is discarded.

For example, a robot wants to pick up an object, and he has several ways of doing so. In trying the first method, with his right hand, he discovers that the object is hot by seriously burning himself.

It is clear that though this method failed he should not go back and try his left hand. Nor should it be necessary for him to have foreseen the difficulty and thus set up a message catcher for burnt hand failure (or for lightning striking); such caution, applied to all possible contingencies, is impossible. The reasonably designed robot will drastically modify his behavior at this point, say by getting a pair of tongs, after screaming.

Notice also that any failure-driven generator (a function that returns a value but sets a failpoint) is constrained to generate alternatives one at a time. If the alternatives are interdependent, surely the best one should be chosen while all or most are in view. In fact, the only reason for <u>generating</u> objects rather than just making a list of them is that sometimes the number of possibilities (as, say, the prime numbers) may be infinite, or the cost of generation of the next possibility is much greater or grows much faster than the cost of testing its usefulness. In many cases, however, an explicit list of all or some of the alternatives is what is desired. Of course, even in PLANNER, such a list must exist, snuggled inside some GOAL's failpoints, but there is no natural way to get at it.

The PLANNER implementation of pattern-directed procedure invocation reinforces these problems of backtracking. The anonymity of the procedures that may be fetched by pattern-directed call makes it even easier to pretend to have many "independent" methods of solving the same

problem, hoping that one of the methods, to be found by failure, will come up with an acceptable solution. Not only does this organization force each method to have to be able to run in complete ignorance of what has been tried so far, or even that other methods exist, but in many cases the "independent" methods will come up with the same unacceptable answer more than once, causing the system to thrash ridiculously. The solution, of course, is to abandon the myth that there could be several independent methods of attacking any interesting problem, and concentrate on techniques for making methods interact reasonably.

This is not to say that the pattern-directed function call does not have its place in the arsenal of problem solving. It is valuable whenever, either due to the infiniteness of the set or the economics of storage vs. computation, a procedure can be used to represent a set of assertions.

There are several such reasonably good ideas scattered throughout PLANNER. They include the notions of "generator" and "possibilities list." But they have been pushed far beneath the surface, so that that the user may think in terms of "goals." While the concept of goal-directedness is certainly as well established as any in our field, it seems clear that naming a primitive function "GOAL" is not enough to capture the essence of the idea. In the next section, we shall concentrate on the decent ideas in PLANNER, and discard those that have gotten so many MICRO-PLANNER users in trouble, starting with automatic

backtracking.

Building CONNIVER

We have shown in the first section that backtracking is a device of questionable usefulness at the very tasks for which it was designed. It encourages a linkage of the mechanism for generating alternatives with the mechanism that restores the environment after the investigation of each one. Each time, the generation of the next must proceed on the basis of very little information besides the fact that the last failed. We have, in the end, a control structure that almost forces the user to regard all his problem-solving methods as independent.

It seems to be the linkage of these two mechanisms in the GOAL statement that is at fault. As an alternative, imagine that we are not allowed to use failure to clean things up, and that everything each goal-directed subroutine does stays done. Then, if the speculation it has indulged in is not to have effects in the environment of its caller (the program considering the alternatives), it must have a local environment of its own to make changes to. These changes may make its model of the problem conflict with its superior's model, or may simply be hypothetical additions to it. The important point is that a simple return to the caller will be sufficient to make the changes invisible.

This concept can be made clear by analogy with the familiar notions of "control environment" (a stack, for example), and "access environment" (where variables are bound; the term is Bobrow and

Wegbreit's[9]); in CONNIVER, we generalize the latter to "data base environment," or context. Just as LISP 1.5 supports a tree of access environments ("association lists"), so CONNIVER supports a tree of contexts, in which each daughter-context represents a data base incrementally different from her parent.

This tree, it will be made clear, must be grown and maintained in conjunction with a control environment of equal complexity. But the control structure exists only to exploit the data base, so we return to it later.

Conniver contexts contain items, which are simple list structures like PLANNER's assertions (without the theorem-proving connotations that surround the latter term). An item such as (SQUARE48 PAWN3) may be added to the current context with

```
(ADD '(SQUARE48 PAWN3))
```

and taken out with

```
(REMOVE '(SQUARE48 PAWN3)).
```

The arguments to ADD and REMOVE are skeletons, list structures that define items after substitution of the values of their variables. Variables are indicated by ",". Thus, if X = PAWN2, (ADD '(SQUARE49 ,X)) adds the item (SQUARE49 PAWN2) to the current context.

Now, if the presence or absence of an item is to be reflected only in a local data base, that is, be "hypothetical," the data

environment must be "pushed down" before doing ADD's and REMOVE's of this sort. Since, in CONNIVER, a context is a data type, and the current context is assigned to the variable CONTEXT, all we need to write is:

```
(PROG "AUX" ((CONTEXT (PUSH-CONTEXT)))
   (ADD '(SQUARE48 PAWN3))
        .
        .
        .   )
```

CONNIVER syntax is roughly that of LISP, but a declaration of local variables must be explicitly indicated with the atom "AUX", and each such local must be given an explicit initial value, if it is not to be unassigned, by being declared as "(variable value)" instead of just "variable." This PROG thus rebinds CONTEXT to the value returned by the system function PUSH-CONTEXT. The current context has had one more context-frame pushed onto it. New changes apply to this frame only. After the body of the PROG has been executed in this "hypothetical" context, the PROG's control frame will be exited. CONTEXT will be unbound, restoring its old value, in which the action of the ADD is invisible; in effect, a data frame has been exited as well.

Since contexts are data structures just like lists, they can be returned as values of functions, assigned to global variables, etc., so that in fact the user has available a tree of contexts his program has left behind, in the same way that using functional arguments (closures of functions) in LISP creates a tree of variable-value associations.

Items can be retrieved from the current context by means of the CONNIVER primitive FETCH, which finds all items present in the context that match a pattern. For example, if we let the presence of the item (HAS player square) means "player (X or O) has put his mark in square in the tic-tac-toe position represented by the current context," we can find all of X's squares with

```
(FETCH '(HAS X ?SQUARE)).
```

Roughly as in PLANNER, the "?" indicates that the variable SQUARE is to be assigned a value by matching the pattern (HAS X ?SQUARE) against some item. However, FETCH does not makes the assignment. Since backtracking has been exorcised from CONNIVER, it simply returns a possibilities list which contains all the matching items, rather than hiding them in a failpoint in GOAL, to be handed to us coyly, one per failure.

Such a possibilities list might look like

```
(*POSSIBILITIES
   (*ITEM ((HAS X 5) (9 +)) ((SQUARE 5)))
   (*ITEM ((HAS X 2) (9 +)) ((SQUARE 2)))   ).
```

The exact meaning of every parenthesis in this list is unimportant, but the overall content is this: FETCH has found two item possibilities, both present in this context (which includes context-frame 9). The first matches the pattern with SQUARE = 5; the second, with SQUARE = 2.

The user can manipulate this list in any way he chooses; one way is with the system function TRY-NEXT, which pops off and returns the first item in the list (here, ((HAS X 5) (9 +))), and assigns the pattern

variables as the possibility directs (and so, in this case, sets SQUARE to 5).

How can we exploit this data structure? Although PLANNER, in its latest incarnations[10], contains a data base structure as powerful as what we have sketched, it should be clear that its backtrack mechanism is worthless for the purpose; it often destroys exactly the contexts we wish to preserve.

For the present, we want to stick to the simple problem of generating alternatives, which PLANNER is constrained to do one at a time, one per failure, erasing part of the generator's context each time. If the generator wishes to communicate the fact that it has failed so far, it must destroy its context entirely to do it, even though it may have just succeeded in building a useful world model. The best that can be said for failure is that it has become irrelevant, since if a process does want to throw in the towel completely, all it has to do is return, unbinding CONTEXT. To use CONNIVER data structures effectively, we need a compromise between keeping a failpoint and returning for good; let us allow a generator to return, but keep its control environment in existence. Since that control environment may include a binding of CONTEXT, it includes some data environment as well; it can be used to embody a point of view towards the program's problem and a place to go to attack it further.

This mechanism becomes important when a program wishes to include a set of items in the current context on the basis of a procedural criterion instead of their actual presence. This is essentially the role of consequent theorems in PLANNER, but the analogous CONNIVER structure, the if-needed method, cannot work the way consequents do because there is no backtracking. An if-needed which matches a FETCH's pattern will be found after all the matching items, and included in the possibilities list as a method possibility, of the form (*METHOD pattern method). When TRY-NEXT encounters such a thing, it must invoke it. An if-needed, once invoked, acts as much like a generator as its consequent-theorem cousin, but in a different way.

Pursuing our tic-tac-toe example, let the presence of the item (WINMOVE player square move) mean "player (X or O) will have three marks in a row if he makes move, after occupying square." Such items might come in handy, for example, in a search for two winning moves after the occupation of square; since the opponent cannot block both, occupying square will force a win.

For example, a CONNIVER program may search for moves M which win the game for X after his occupation of square 8 with (FETCH '(WINMOVE X 8 ?M)). If the resulting possibilities list includes (*METHOD (WINMOVE X 8 ?M) WINMOVES), the method WINMOVES will be invoked by TRY-NEXT:

```
(IF-NEEDED WINMOVES
      (WINMOVE ?PLAYER ?SQUARE ?MOVE)
      "AUX" (PLAYER SQUARE MOVE (CONTEXT (PUSH-CONTEXT)) P1 SQ1 P2 SQ2)
      (ADD '(HAS ,PLAYER ,SQUARE))
      (REMOVE '(FREE ,SQUARE))
      (CSETQ P1 (FETCH '(HAS ,PLAYER ?SQ1)))
   :OUTERLOOP
         (TRY-NEXT P1 '(GO 'END))
         (CSETQ P2 (FETCH '(HAS ,PLAYER ?SQ2)))
   :INNERLOOP
            (TRY-NEXT P2 '(GO 'OUTERLOOP))
            (COND ((AND (LESSP SQ1 SQ2)
                        (CSETQ MOVE (THIRD-IN-ROW SQ1 SQ2))
                        (PRESENT '(FREE ,MOVE)))
                   (NOTE (INSTANCE)))   )
            (GO 'INNERLOOP)
   :END
      (ADIEU)   ).
```

When WINMOVES is invoked, it pushes CONTEXT down, and supposes that PLAYER owns SQUARE, and that it is no longer free. The two nested, TRY-NEXT-driven loops then consider each pair of squares owned by PLAYER, setting SQ1 and SQ2 at statements :OUTERLOOP and :INNERLOOP, respectively. (Atoms used as labels must be prefixed with the character ":".) The second argument to each TRY-NEXT is evaluated when its possibilities are exhausted. THIRD-IN-ROW is a function that returns the third square in the row, column, or diagonal of its arguments, or NIL if they are not collinear; (PRESENT pattern) is non-NIL only if an item matching pattern is present in the current context.

Here is how WINMOVES works: for each distinct pair of collinear squares owned by PLAYER, if the third square is free, the INSTANCE formed by substituting the current value of MOVE into the pattern used to call WINMOVES is NOTEd. (I.e., it is saved on a list, accessible to the user,

whose structure need not concern us here.) When all the instances (which look just like item possibilities) have been found, they are packaged by (ADIEU) into a possibilities list, which is returned. It might look like

```
(*POSSIBILITIES
   (*ITEM ((WINMOVE X 3 5)) ((M 5)))
   (*ITEM ((WINMOVE X 3 3)) ((M 3)))   ).
```

The list which ADIEU creates is returned to the TRY-NEXT that invoked WINMOVES. This TRY-NEXT has been manipulating our original possibilities list, generated by (FETCH '(WINMOVE X 3 ?M)); it found WINMOVES in the list and invoked it, and it now has its value, a new list of possibilities. It does the obvious: it splices the list the method returned into its list in place of WINMOVES. Thus, the original generator possibility in the list has been made to stand for the possibilities it can produce when invoked. WINMOVES stands for a set of winning moves not mentioned explicitly in the data base.

Our concept of generator appears simpler than PLANNER's; WINMOVES dumps all the instances into the upper possibilities list and returns, leaving its control environment and binding of CONTEXT to be collected as garbage. Even if its caller wants only one new item possibility, generators like WINMOVES give him all of them.

We have returned to our original problem: how can we maintain in existence the control and context structure of WINMOVES while returning from it with only a few of the possibilities it can find? The answer

lies in the structure and function of the possibilities list; to invoke a method found in such a list is to replace the method by its value, itself a list of possibilities. If this value list contains a generalized tag back to the generator's activation, its environment will be preserved. Not only that, but if TRY-NEXT comes upon such a thing in a possibilities list, it is bound to GO to it. Now the method can generate items in finite groups, asking to be reawakened if none of the items satisfies its caller. A new version of WINMOVES that works this way (and has been streamlined in other respects) looks like:

```
(IF-NEEDED WINMOVES
      (WINMOVE ?PLAYER ?SQUARE ?MOVE)
      "AUX" (PLAYER SQUARE MOVE (CONTEXT (PUSH-CONTEXT)) SQ1 SQ2)
      (ADD '(HAS ,PLAYER ,SQUARE))
      (REMOVE '(FREE ,SQUARE))
      (FOR-EACH (FETCH '(HAS ,PLAYER ?SQ1))
         (FOR-EACH (FETCH '(HAS ,PLAYER ?SQ2))
            (COND ((AND (LESSP SQ1 SQ2)
                        (CSETQ MOVE (THIRD-IN-ROW SQ1 SQ2))
                        (PRESENT '(FREE ,MOVE)))
                   (NOTE (INSTANCE))
                   (AU-REVOIR))   )   ))
      (ADIEU)   ).
```

Aside from the use of FOR-EACH as a shorthand for a TRY-NEXT-driven loop, the only addition is (AU-REVOIR) following (NOTE (INSTANCE)). AU-REVOIR is just like ADIEU, but adds a tag to its own activation at the end of the possibilities list it returns. Now WINMOVES NOTEs and returns just one instance each time, but if the instance is unsatisfactory, is reawakened at the end of the inner FOR-EACH, to generate one more the same way. (Note that on such returns to an activation, it is the tag AU-REVOIR left in the upper possibilities list that stands for a list of new items; GOing replaces it with a possibilities list from the generator

just as invoking does with method possibilities.)

The requirement that there be generalized tags, tags that mention whole control environments, makes it necessary that CONNIVER maintain a control tree similar in structure to the context tree it serves. All such still-viable environments form a set of processes cooperating to solve a problem. Some of these are generators, using possibilities lists as communication channels with their callers, but this by no means exhausts the alternative ways of interacting. In particular, CONNIVER's generalized control structure makes it easy to put all of failure and backtracking back in if the user wants them, but he has the duty (or privilege) of designing and maintaining control over what he builds.

A couple of points remain to be made. Notice that, although the loops in WINMOVES are exhaustive and blind, they are explicit. The only natural way to write this generator in PLANNER is by use of successive GOAL statements that filter out the bad choices. Although the user may intend a loop like a FOR-EACH, and, locally, the GOAL conglomeration behaves like one, it suffers uncontrollably from the effects of global failures.

Generators do not have to be methods; we have only been pursuing this example because of the PLANNER analogy; it seems much more rational that WINMOVES in particular be a function of two arguments, PLAYER and SQUARE, with values corresponding to MOVE. (See [3].)

Communication between processes, it is our feeling, is essential to their success. We have tried to build as many communication devices into TRY-NEXT and generators as possible in hopes that they will be used. It would be very dangerous to try extending the exhaustive searches used in WINMOVES to something as much more complicated as a plausible chess move generator. A clever generator must be able to talk to its caller. WINMOVES is supposed to illustrate what is legal in CONNIVER, not what is good.

We have constructed CONNIVER partly by raising to prominence ideas casually embedded in PLANNER, partly by giving hidden PLANNER constructs back to the people, and partly by concentrating on what is needed in a programming language as opposed to a theorem-prover. Our major contribution, we think, is the elimination of backtracking upon failure as a mechanism for the blind generation of alternative approaches to a problem. We have shown how PLANNER makes it difficult to write controllable programs; how, like most theorem-provers, it is committed to loosely guided exhaustive search as a problem-solving method; and how the user must either succumb to the will of the control structure or spend much of his time using primitives (like FINALIZE, STRAIGHTEN, TEMPROG, etc. ad infinitum) that save him from it. It is our hope that we have shown that control and understanding of his programs should be vital concerns of the Artificial Intelligence programmer.

References

1. Hewitt, C. (1970)
PLANNER: A Language for Manipulating Models and Proving Theorems in a Robot
MIT, AI Memo 168 (Revised); see also Walker, D.E. and Norton, L.M., eds., Proc. IJCAI 1, pp. 295-301

2. Sussman, G.J., Winograd T., Charniak E. (1971)
MICRO-PLANNER Reference Manual
MIT, AI Memo 203A

3. McDermott, D., Sussman G.J. (1972)
The CONNIVER Reference Manual
MIT, AI Memo ***

4. McCarthy, J. et. al. (1962)
LISP 1.5 Programmer's Manual
Cambridge, MIT Press

5. Moses J., (1967)
Symbolic Integration
MIT, Cambridge, Mass., Ph.D. dissertation

6. Slagle, J., (1961)
A Computer Program for Solving Problems in Freshman Calculus (SAINT)
MIT, Cambridge, Mass., Ph.D. dissertation; also in Feigenbaum, E.A. and Feldman, J. eds., Computers and Thought (McGraw-Hill), pp. 191-203

7. Winston, P.H., (1971)
Heterarchy in the MIT Robot
MIT, AI Vision Flash 8

8. Greenblatt, R., et. al., (1967)
"The Greenblatt Chess Program,"
Proc. FJCC, pp. 801-810; also MIT (1969), AI Memo 174

9. Bobrow, D.G., Wegbreit, B., (1972)
A Model and Stack Implementation of Multiple Environments
Bolt Beranek and Newman Inc. Report No. 2334

10. Hewitt, C. (1972)
Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot
MIT, Revised Ph.D. dissertation, AI Technical Report - 258

WHY CONNIVING IS BETTER THAN PLANNING

by

Gerald Jay Sussman and Drew Vincent McDermott

Massachusetts Institute of Technology
Artificial Intelligence Laboratory
Cambridge, Massachusetts

Gerald Jay Sussman
Room 818
545 Technology Square
Cambridge, Mass. 02139