PLANNING AND IMPLEMENTATION ERRORS IN ALGORITHM DESIGN

Wayne D. Gray,
U. S. Army Research Institute

Albert T. Corbett, & Kurt Van Lehn
Carnegie-Mellon University

Introduction

This study examines the algorithm design process for 59 LISP programmers who tackle a classic artificial intelligence search problem for the first time. Programmers were asked to code a single function called *descendent*, that was of average length and that performs a depth-first search over an hierarchy. This was a fairly difficult task. In this paper, we outline a set of basic planning steps for designing this algorithm and examine variations in the 59 solutions that reflect divergences at different steps.

Various aspects of algorithm and software design have been studied before and this study owes much to those efforts (for example, 1, 3, 8, 10, and 14). Especially relevant are those studies that have emphasized "bugs" (9, 12, 13).

The current study looks for evidence of **plan** or **implementation** failures (bugs) at each step in the design process. Plan failures are indicated by the use of plans inappropriate to the current problem (5) and are a type of negative transfer (7). Implementation failures result from the failure to correctly translate a plan into the programming language (11).

The Study.

The Programmers.

Fifty-nine students were drawn from four LISP courses and one course on cognitive science. All students used the same introductory LISP textbook (2) and completed at least those lessons in the LISP Tutor (4) that covered the functions and basic control structures required in this study. At the time of the study, no student had attended lectures or read the textbook chapter on search techniques.

The Problem Specification

Programmers were asked to write a depth-first search[1] function that took two arguments and determined whether the second argument was a descendent of the first. If so, the function returned "**t**," otherwise "**nil**." The problem statement was accompanied by the hierarchy in Figure 1. Given this example, ***(descendent 'Bill 'Frank)*** should return ***t***, while ***(descendent 'Bill 'Joe)*** should return ***nil***.

Two important constraints were imposed upon the task. First, programmers were not given any information about how hierarchies were represented in LISP. Their only means for searching hierarchies was an expansion function (called ***expand***) that accepted one node and returned a list of the node's immediate descendents. For example, from Figure 1 ***(expand 'Bill)*** would return ***(Julia Mike)***. Second, programmers were to write an iterative function with no recursive function calls. (More specifically, they were required to use a **let/loop** construction rather than **do**, to further standardize the goal state).

Finally, the problem description recommended that a local variable be used to save the list of nodes that was generated by the expansion function until they could be checked. The following function definition satisfies these constraints:

[1]A depth-first search of a tree moves down whenever possible to get the next node, and only moves back up and over when it is not possible to move down. A depth-first search of the tree in Figure 1 might check the nodes in the following order: Harry, Jane, Joe, Diane, Bill, Julia, Frank, Anne, Susan, Mike.

```
(defun descendent (ancestor target)
  (let ((queue (list ancestor)))
    (loop
      (cond ((null queue) (return nil))
            ((equal (car queue) target) (return t)))
      (setq queue (append (expand (car queue)) (cdr queue))))))
```

In this function, ***queue*** holds a list of nodes that have been accessed and need to be checked. Each time through the loop ***queue*** is tested. If it is empty, the search has failed and the function returns ***nil***. If it is not empty, the first node in ***queue*** is compared to the target and if they match the function returns ***t***. Otherwise, the first node is removed from ***queue*** while its immediate descendents are added to the list. In the case of Figure 1, if ***queue*** held the list ***(Jane Bill)*** and the first node ***Jane*** was tested and rejected, then on the next cycle ***queue*** would hold ***(Joe Diane Bill)***.

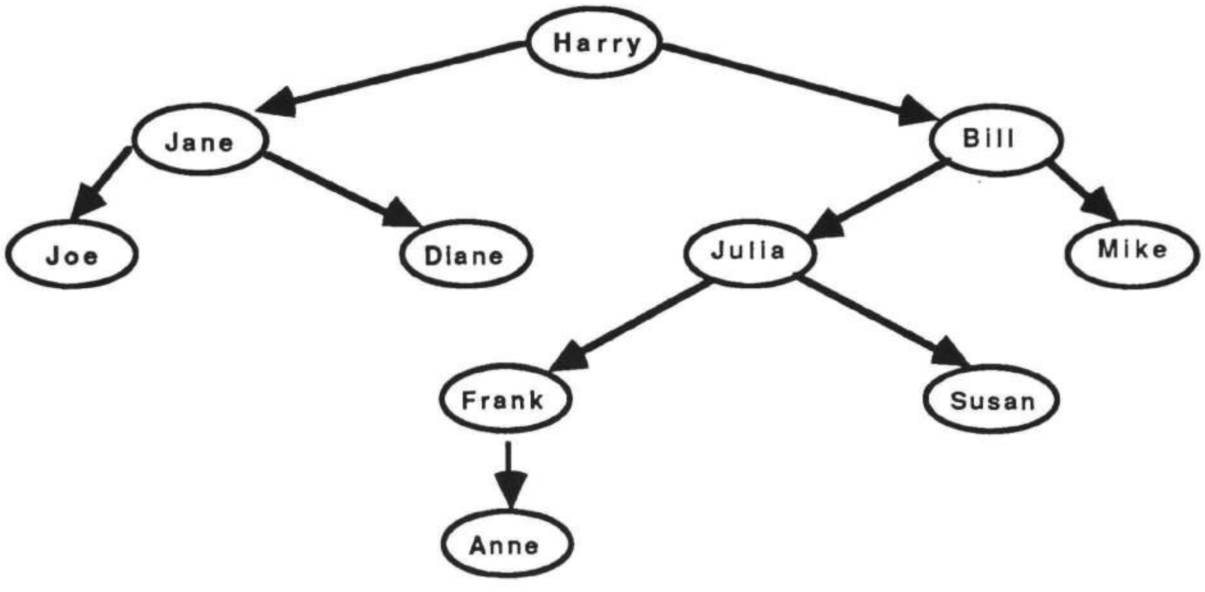

Figure 1
Example Hierarchy given to programmers coding "descendent."

Algorithm Design

At first glance, the solution consists of a simple two step cycle:[2] 1. ***Get the Next Node***, 2. ***Test the Node***. The second step is easy to implement, but the ***get the next node*** step is quite difficult. If programmers were given a function that takes a node and simply returns the appropriate next node to test, then this problem becomes trivial. Instead, the function, ***expand***, accepts one node and returns a ***list*** of all the descendents of that node.

expand imposes a constraint on the solution that is not an intrinsic part of a depth-first search over a tree. The function returns a list on each cycle rather than directly accessing individual nodes in the tree. This expansion function constraint leads most directly to an algorithm that is essentially recursive. If we label the algorithm ***CHECK-LIST***, we can represent its recursive structure as the sequence of operations shown in Figure 2.

[2]There are, in fact, additional issues the student must address before completing the design. For example, each cycle must also contain a test to see if the network is exhausted.

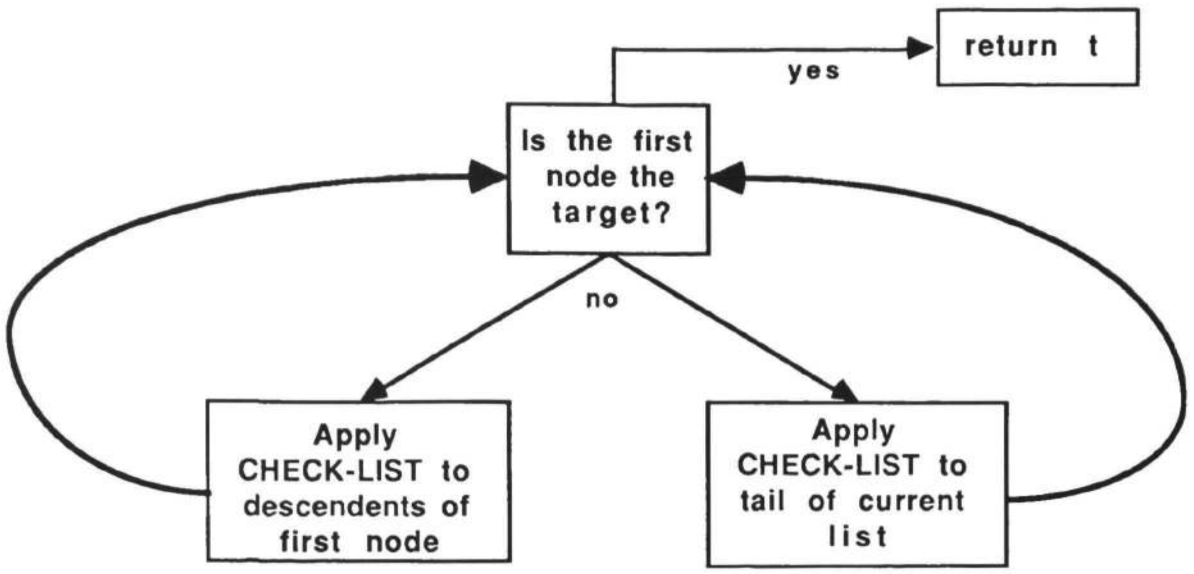
return t
yes
Is the first node the target?
no
Apply CHECK-LIST to descendents of first node
Apply CHECK-LIST to tail of current list

Figure 2
Simplified Recursive Solution

The corresponding LISP function might be coded as follows:

```
(defun descendent (given target)
    (check-list (list given) target))
(defun check-list (current-list target)
    (cond ((null current-list) nil)
          ((equal (car current-list) target) t)
          (t (or (check-list (expand (car current-list)) target)
                 (check-list (cdr current-list) target)))))
```

This solution conforms to a type of recursion, ***car-cdr*** recursion, with which the programmers were familiar. The iterative constraint blocked this solution, of course, and sets up the most demanding aspect of the planning process.

Students are required to discover an iterative solution which is isomorphic to this recursive concept. That isomorphic iterative algorithm can be specified as shown in Figure 3. This specification gives rise to the definition of descendent presented earlier.

While the results of the iterative solution are isomorphic to the recursive solution, there is an important conceptual difference. The recursive solution does not require building a new list structure, the iterative solution does. In Figure 2, the recursive function, **CHECK-LIST**, is applied to both the existing list structure and the list structure returned by ***expand***. In contrast, the iterative solution requires building a new list on each iteration. As shown in Figure 3, on each cycle, two different operations are performed upon the list and the results of these two operations are combined into a new list. (For this reason the iterative solution will be referred to as the ***list-building*** algorithm.)

This characterization of the iterative solution suggests three steps in the design process that may cause difficulties and lead to bugs in the students' code: (1) the transition from thinking in terms of individual nodes to thinking in terms of expansion lists, (2) the recognition that a depth-first search requires a solution analogous to ***car-cdr*** recursion, and (3) the recognition that, unlike the recursive solution, the iterative algorithm requires that a new list be built on each cycle.

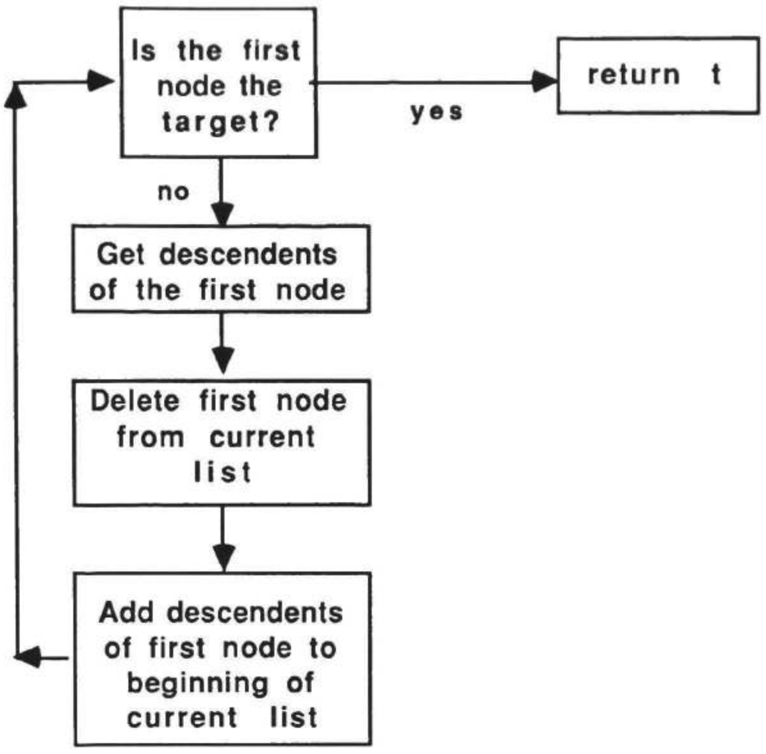
Is the first node the target?
yes
return t
no
Get descendents of the first node
Delete first node from current list
Add descendents of first node to beginning of current list

Figure 3
Simplified Iterative Solution

Simulating the algorithm.

A subset of 23 programmers were asked to perform a paper and pencil simulation of the function prior to coding it. They were asked to simulate the function call ***(descendent 'Harry 'Frank)***, by writing down the initial value of the variable ***queue*** and then, for each iterative cycle, writing the node that would be checked, the immediate descendents of the node, and the new value of ***queue***. Thus, for one cycle, the to-be-checked node is ***Jane***, its immediate descendents are ***(Joe Diane)*** and the new value of ***queue*** is ***(Joe Diane Bill)***. Programmers were given feedback to ensure that they simulated the function correctly. Programmers who were guided directly through the iterative solution would not be expected to conform to the standard algorithm design sequence proposed in the earlier section.

Procedure

Programmers were asked to *talk-aloud* while coding the function, that is, to report what they were thinking as they worked (6). At the beginning of the session, each programmer was given practice in talking aloud and then read an abridged version of the search chapter from their LISP textbook that described hierarchies, depth-first search, and expansion functions. This abridged version did not discuss how to implement a search function.

After reading the text, the programmer was given the problem description and asked to write down the order in which the nodes should be checked, to ensure that s/he understood the concept of a depth-first search. Then the programmer simulated the function if s/he were one of the 23 programmers in the simulation condition. Finally, the programmer coded the function on a computer terminal while talking aloud. The programmers worked on the function until either (1) they were satisfied with their solution or (2) gave up on it or (3) one hour elapsed. Programmers were not able to test their function in the course of coding it.

Results and Discussion.

This was a relatively difficult task. Of the 59 programmers, only 3 wrote functions that would work with no modifications. Thirty-five solutions contained only minor implementation[3] and/or planning errors, 17 contained major planning errors, but only 4 were completely uninterpretable. Viewed to emphasize the positive, 55 of the 59 programmers wrote functions that contained an interpretable control structure that in principle would have searched some or all of the tree. The interesting question, however, is not how many programmers coded the function correctly, but what the final code reveals about difficulties in planning and implementation.

Table 1
Comparison of Algorithm Use by Type of Training

	simulation	control	totals
list-builders	19	18	37
other	1	15	16
totals	20	33	53

Fifty-three of the 59 solutions were rated as trying to solve the correct problem.[4] Of these 53, 38 apparently worked through the planning process successfully. (Of these, 37 programmers generated solutions that are consistent with the *list-building* solution described above and one designed a unique solution consistent with the problem statement.) The simulation manipulation reveals whether the basic difficulty is in generating the plan or implementing it. Of the 20 programmers who simulated an example of the list-building algorithm, 19 (95%) generated code that conformed to the algorithm. Only 18 of 33 (55%) programmers in the control condition conformed to the list-building algorithm. The results are shown in Table 1. A chi-sq test showed these differences to be significant (Chi-sq [1, **n**=53] = 7.85) ($p < .05$), indicating that the simulation trained programmers used the list-building algorithm more often then would be expected and suggesting that when a solution deviated from the algorithm it was largely because of difficulty in generating the plan, rather than an implementation failure.

Planning Steps: Node Testing.

As described above, the node test is easy to plan and implement and all but one solution contained a test that compared the nodes in the tree to the second argument. However, there is an interesting difference among the 53 programmers who wrote interpretable code. Rather than testing a single node on each cycle, (using ***equal***), 14 of the programmers compared the target to a whole list of nodes (using ***member***).

[3]Implementation errors include both syntactic and semantic errors. Examples of minor syntactic errors include misplace or missing parentheses, or inappropriate use of quotes. Examples of minor semantic errors include substituting a similar, but inappropriate LISP function for the correct one (such as using ***cons*** for ***append*** in the update) or initializing the local variable to a node (a LISP atom) when it should have been initialized to a list containing the node.

[4]We could make no sense of 4 solutions, so these were eliminated from further consideration. Likewise, we have not included two unique, but non-depth-first search algorithms. These exclusions leave 20 programmers with simulation training and 33 without for a total of 53.

We hypothesize that the choice ***member*** versus ***equal*** represents a planning, not an implementation, bug. Our programmers were very familiar with both ***member*** and ***equal***. At the implementation level it seems unlikely that one would be mistaken for the other. In contrast, the use of ***member*** may represent the transfer of a very natural perceptual strategy. If we were to physically retrieve a list of descendents (especially a short list), it would be nearly impossible not to scan the entire retrieved list and determine if the target is on the list. We hypothesize that the use of ***member*** is evidence that this *naive* plan has substituted for the node test plan required by the problem specification.

Planning Steps: Getting New Nodes.
The nature of the expansion function imposes constraints on the algorithm for getting new nodes. In particular, it imposes a list structure on the planning process and gives rise most naturally to a recursive solution. The ban on recursive function calls constrains the programmer to transform the recursive solution into an iterative solution that builds a list.

There is some evidence that a few programmers had difficulties with superimposing a list structure on the tree diagram. Specifically, 6 programmers generated solutions in which a local variable was processed in some contexts as if it stored a single node and in other contexts as if it stored a list. The remaining 47 programmers did not appear to have this difficulty.

Thirty-seven programmers employed the list-building iterative solution in generating new nodes. One programmer employed LISP *property lists* to generate a hierarchical structure that directly paralleled the diagrammatic tree structure and used these properties to structure the search process. This solution is fascinating since it diverges widely from the standard plan and hints at the actual size of the algorithm space. It will not be considered further, precisely because it does not cast light on the difficulties of the list-building plan. The final two categories represent fundamentally flawed variations of the list-building plan.

The first variation, coded by seven of the programmers, is a ***depth-first/dead-end*** search of the tree. In this algorithm, the first element in ***queue*** is searched and expanded in each cycle. However, the remainder of ***queue*** is discarded and ***queue*** is set equal to the expansion, as in the following LISP expression which would be substituted for the final line of the list-building solution:

```
(setq queue (expand (car queue))).
```

This solution searches down one branch of the tree (if the target is not found along the way) and then terminates. In the case of Figure 1, the nodes ***Harry***, ***Jane***, and ***Joe*** would be checked.

The second variation, coded by 7 different programmers, might be called a ***two-step*** algorithm. In this algorithm, ***queue*** is initialized to the expansion of the first argument. Then in every cycle the first element in ***queue*** is expanded, that expansion is tested (with a **member** test), and the element is removed from ***queue***. The following code, which would be substituted for the final two lines of the list-building solution, characterizes this algorithm:

```
(cond ((member target (expand (car queue))) (return t)))
(setq queue (cdr queue))
```

This solution will search the top two levels of the tree. At least some of the 7 programmers recognized that the solution was inadequate and tried to extend it with baroque yet futile additions, for example, by incorporating an inner loop to reach down another ply in the tree.

These categories are interesting in that the failure is closely linked to different aspects of the plan described earlier. The ***depth-first/dead-end*** solution may represent a failure to fully formulate the recursive plan. That is, this solution checks and expands the first node in the queue in each cycle, much as each call to a recursively defined function would, but completely fails to process the tail of the list, in effect omitting the ***cdr*** component of the ***car-cdr*** recursion.

The ***two-step*** approach, on the other hand, seems to be based on a fully specified recursive solution that is not correctly translated into an iterative solution. In this solution, the processing of the tail of the list is structurally correct, as is the expansion of the *car*. However, the requirement to build a new list on each iteration is not recognized. This solution may directly reflect the programming experience of the programmers. All students learned how to code equivalent *tail recursive* and *list iteration* functions, and this component of the algorithm is coded correctly. On the other hand, while students also encountered *car-cdr* recursive functions, this experiment was their first experience in generating equivalent iterative functions.

Conclusion

This report is necessarily brief and by omitting discussion of various issues concerning both systematic and non-systematic deviations may not fully convey the degree of variability obtained across solutions in this experiment. Moreover, there remain response patterns that are difficult to evaluate simply on the basis of the final code. For example, some programmers test whether a node has descendents before adding the descendents to the queue although, given the definition of the function ***append***, this step is unnecessary. It is unclear from examining the solutions whether this is a plan bug imported from naive notions of hierarchical search (see also 5), or an implementation bug tacked on because of uncertainty about how ***append*** works. The long term goal of this research is to develop a more detailed model of the algorithm design process, on the basis of keystroke data and tapes of the coding sessions.

Nevertheless, granting the wide degree of coding variability obtained in this study, it is possible to discern categories of errors that reflect not just implementation failures, but failures in predictable steps in the algorithm design process

References

1. Adelson, B., & Soloway, E. (1987). A model of software design. In M. Chi, R. Glaser, and M. Farr (Eds.), **The nature of expertise.** Hillsdale, NJ: Erlbaum.
2. Anderson, J. R., Corbett, A. T., & Reiser, B. J. (1987). **Essential LISP**. Reading, MA: Addison-Wesley Publishing Company, Inc.
3. Anderson, J. R., Farrell, R., & Sauers, R. (1984). Learning to program in LISP. **Cognitive Science, 8**, 87-129.
4. Anderson, J. R., & Reiser, B. J. (1985, April). The LISP tutor. **Byte**, pp.159-175.
5. Bonar, J., & Soloway, E. (1985). Pre-programming knowledge: A major source of misconceptions in novice programmers. **Human-Computer Interaction, 1.**
6. Ericsson, K. A., & Simon, H. A. (1985). **Protocol analysis: Verbal reports as data**. Cambridge, MA: MIT Press.
7. Gray, W. D., & Orasanu, J. (1987). Transfer of cognitive skills. In S. Cormier and J. Hagman (Eds.), **Transfer of learning**. Orlando, FL: Academic Press.
8. Jeffries, R., Turner, A. A., Polson, P. G., & Atwood, M. E. (1981). The processes involved in designing software. In J. R. Anderson (Ed.), **Cognitive skills and their acquisition.** Hillsdale, NJ: Erlbaum.
9. Johnson, W. L., & Soloway, E. (1984). Intention-based diagnosis of programming errors. **Proceedings of the 1984 Conference of the AAAI**, 162-168.
10. Kant, E. (1985). Understanding and automating algorithm design. **IEEE Transactions on Software Engineering, 11**, 1361-1374.
11. Moran, T. P. (1983). Getting into a system: External-internal task mapping analysis. In **Proceedings of the ACM SIGCHI Conference on Human Factors in Computer Systems**. Boston, MA.
12. Soloway, E. (1985). From problems to programs via plans: The content and structure of knowledge for introductory LISP programming. **Journal of Educational Computing Research, 1**, 157-172.
13. Soloway, E., Bonar, J., & Ehrlich, K. (1983). Cognitive strategies and looping constructs: An empirical study. **Communications of the ACM, 26**, 853-860.
14. Steier, D. M., & Kant, E. (1985). The roles of execution and analysis in algorithm design. **IEEE Transactions on Software Engineering, 11**, 1375-1386.