CMU-CS-82-145

(12)

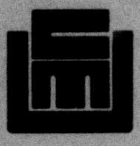
Problem Solving Techniques for the Design of Algorithms

Elaine Kant and Allen Newell

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

23 November 1982

DEPARTMENT
of
COMPUTER SCIENCE



Carnegie-Mellon University

CMU-C S-82-145

Problem Solving Techniques for the Design of Algorithms

Elaine Kant and Allen Newell

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

23 November 1982

This paper will appear in *Information Processing and Management.*

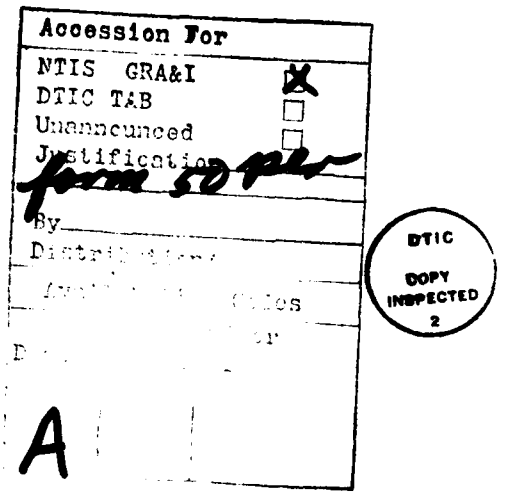
Abstract

By studying the problem-solving techniques that people use to design algorithms we can learn something about building systems that automatically derive algorithms or assist human designers. In this paper we present a model of algorithm design based on our analysis of the protocols of two subjects designing three convex hull algorithms. The subjects work mainly in a data-flow problem space in which the objects are representations of partially specified algorithms. A small number of general-purpose operators construct and modify the representations; these operators are adapted to the current problem state by means-ends analysis. The problem space also includes knowledge-rich schemas such as divide and conquer that subjects incorporate into their algorithms. A particularly versatile problem-solving method in this problem space is symbolic execution, which can be used to refine, verify, or explain components of an algorithm. The subjects also work in a task-domain space about geometry. The interplay between problem solving in the two spaces makes possible the process of discovery. We have observed that the time a subject takes to design an algorithm is proportional to the number of components in the algorithm's data-flow representation. Finally, the details of the problem spaces provide a model for building a robust automated system.

This research is supported by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

A

Table of Contents



List of Figures

1. Algorithm Design and Software Science

Software, as everyone knows, is expensive to build and maintain. One approach to the problem of generating the large volumes of software being demanded is to automate its production. There are many different types of software and many different phases in software development, and the type of automation tool that is appropriate varies accordingly. In this paper we discuss one of the activities in software development -- algorithm design. This activity typically occurs after the decomposition of a large system into modules and before the more straightforward coding processes to be accomplished by programmers or automatic programming systems. It involves transforming a declarative statement of *what* is to be done into a procedural specification of *how* to do it. Of particular interest here is the use of psychological knowledge to aid in the design of software tools.

Algorithm design, as practiced by the computer scientist, is an activity requiring a great deal of knowledge and intelligence. Although there are both theories and analyses for simpler synthesis tasks, algorithm design is substantially more advanced than anything accomplished to date. It would be quite useful to understand the types of problem solving that occur during design. Our approach to understanding this problem solving involves analyzing protocols from sessions with people designing algorithms. Based on these protocols, we have developed a model of the human problem-solving process involved. In this paper we present our model and consider some lessons for automating the design process (see also [11]). First we discuss how algorithm design might be automated and the possible benefits of studying human design techniques. In Section 2 we present our methods for studying algorithm design and in Section 3 we give some case studies of two subjects designing three algorithms for the same task. We then (Section 4) summarize our model and discuss the role of discovery in problem solving. Finally, we compare the different designs of the case study according to our model (Section 5) and discuss some conclusions (Section 6).

1.1. Automation methods for algorithm design

Although little work has been done on high-level, creative algorithm design, there has been some related research on program synthesis and algorithm optimization that suggests some approaches for automating algorithm design.[1] Possible automation techniques include:

- program transformation (based on expert knowledge)
- formal derivation
- inductive learning systems

---

[1] We are not concerned here with automation involving requirements languages or configuration management systems for large collections of simple components and interfaces.

Much of the existing work on program synthesis that comes closest to algorithm design [1, 4, 8, 23] uses successive refinement by program transformation as a basic organizing principle. Program synthesis involves implementing a program from a very high level specification, but not automatically deriving an algorithm. These program synthesis systems usually focus on selecting data structures or applying user-specified transformations to develop an algorithm. The transformations modify program sections in a variety of ways based on expert knowledge about programming. However, these systems are rather brittle because they require that all details about programming be specified in advance; they do not have any general problem-solving knowledge and do not learn.

Formal derivation systems apply only a small set of transformations that expand definitions of recursion equations, recognize instances of expansions, replace them by function calls, and accomplish a few other similar, general tasks. Such approaches have been used to synthesize sorting algorithms [5] and list-copying algorithms [14]. One problem not addressed by most formal derivation systems is how the axiom sets and applicable transformation rules are chosen; human intervention is usually required to provide the creativity necessary to specify an appropriately limited axiom set and interesting auxiliary definitions. The LOPS system [2] does address this problem by guessing recursive solutions to problems specified in logical equations and by verifying or modifying the guesses with the aid of both a theorem prover and a small model constructed from the axioms. Another variant of the formal derivation approach with some affinities to the present work, [26], involves applying rules for moving constraints across and into generators rather than the application of the standard transformation rules.

Inductive learning of procedures from examples of input-output pairs has thus far only been applied to simple problems. This approach usually involves matching to schemas and heuristic search (for example [25]). Induction from traces has also been studied (for example [22]). If a problem solution can be inferred from watching a person solve a particular example, then induction based on traces may be appropriate. These techniques have been around for some time and so far have shown little signs of evolution.

Much of the activity of software construction discussed above is more routine than the algorithm design problem to be described here. Some varieties of software construction are already well enough understood to be totally automated without any need to investigate how people perform the same tasks. For example, several strategies for data structure selection have been suggested [10, 16, 18, 24]. Also, formal derivation techniques work when a problem is well specified and straightforward optimizations are required.

1.2. Why study how people design algorithms?

We are interested in human problem solving and design strategies for many reasons. First, we do not yet understand how to automate the more difficult parts of design, so studying how people develop complex algorithms shows us one possible approach.[2] Second, a system with an organization similar to that of a human being allows the use of people as expert sources of techniques for getting the system started and as resources to be examined as the system evolves. Third, since the human system organization is one that permits continual augmentation and adaption, this approach may lead to an automatic system that could eventually learn some design principles on its own. Fourth, from what we already know of human behavior, the mechanisms and representations will be flexible and robust, properties sorely needed by current systems. Fifth, it is useful to understand how people think about design. A design assistant program that can follow suggestions to carry out routine subtasks, act as a sounding board, give advice to moderately skilled programmers, or teach novices, requires such understanding. Even the simpler, possibly automatically designed parts of a system may have to be explained or modified or interfaced to human-designed parts. Thus, there are many putative advantages to studying how humans design. The real issue is whether useful knowledge can be obtained.

2. A Method for Studying Algorithm Design

2.1. The problem space theory

Our analysis is driven by a theory of how humans solve problems that has wide currency in cognitive psychology [20, 21]. The central concept is the *problem space*. A problem space contains partial knowledge about a problem and its solution (the *current state*). The subject has a set of *operators* that can be applied to the current state to produce a new state. The subject starts with an *initial state* (here, the problem as posed by the experimenters) and tries to discover a state that contains a solution (here, an algorithm). Behavior in this space involves a *search*, since the subject usually does not have enough knowledge to proceed directly to the final desired state, especially if the problem is difficult. The subject does, of course, have some (often substantial) *search control knowledge* that guides the selection of which operators to apply. But in general the subject will try various paths and run out false leads into dead ends, causing a return to earlier states that can be remembered (or constructed), and eventually will proceed down more appropriate paths. The current state grows throughout the problem solving, as the subject gradually explores the space and acquires knowledge of its various aspects.

---

[2] There has been one interesting study about how people decompose complex programming problems [9].

More than one problem space can be created during problem solving. Satisfying subgoals may imply working in the same space or may require an entirely different space. For instance, if the main space is one of algorithms for the convex hull, as in our task, a problem about the geometry of points in the plane must be settled in a space whose elements are point sets, not algorithms.

2.2. The role of protocol analysis

The problem space theory says that people design algorithms by searching in problem spaces. To use that knowledge to help us design an algorithm discovery system, we need to find out what the problem spaces of the subjects are -- what representations and operators exist and what search-control knowledge guides their search. Given such details, we can expect some strong hints about algorithm discovery systems.

The appropriate experimental technique to answer these questions [21, 7] is to set qualified subjects some tasks of discovering algorithms and to have them talk aloud while working. We record a detailed *protocol* of their solving behavior, and then analyze this behavior in detail. The analysis of the protocol involves hypothesizing problem spaces and showing by detailed analysis of the moves that the subject makes and the information mentioned that these are indeed the correct spaces. This requires specifying the states of the spaces, the operators, and the search control (i.e., operator selection heuristics, state evaluations, goals, and methods). The same total body of evidence (the protocol) is used both to induct the problem spaces and to test them (indeed, the analysis is highly iterative). Thus, the evidence comes from the web of detailed agreement and the initially obscure comments of the subject that make sense given the final posited problem space organization. We do not attempt to simulate the full protocols or understand all the steps the subjects take, but we must specify the results we take from the analysis well enough to be interesting for potential systems.

2.3. The issues

Our initial protocol analysis (described in [11]) focused on understanding the problem spaces that our first subject used in the first segment of the design session. We identified two problem spaces, an algorithm design space and a task-domain space (a geometry space). The initial issues of interest were how to represent partial knowledge and problem-space operators and how to apply operators and control search. To address these issues, we constructed a detailed trace of the behavior of the subject in the main algorithm-design problem space, showing the goals and subgoals, the operator applications, and the search control knowledge used to make various selections and evaluations.

This resulting initial model of algorithm design enabled us to analyze some additional protocols much more quickly than our original attempt. We also analyzed, in less detail, the problem solving in

much more quickly than our original attempt. We also analyzed, in less detail, the problem solving in the task-domain space. This work revealed some new issues and allowed us to compare several design sessions. While we are interested in the details of the subjects' problem spaces and how they fit together (the exact representations and operators that the subjects are using) because the spaces are major candidates for incorporation into an automated algorithm designer, the analyses are too long to present here. Instead, we summarize the general approach we have identified, concentrating on our new findings about the search control mechanisms and about the process of making discoveries. We also compare the design sessions.

2.4. The problem domain

For this study, computational geometry, and in particular the convex-hull construction problem, has been chosen as a realistic design domain. One advantage of this domain is that people have reasonable intuitions about geometry, so the problem is easily explained and subjects need not have specialized backgrounds in computational geometry. Since the algorithms for convex hulls are not well known, it is possible to find naive but intelligent and theoretically sophisticated subjects who can concentrate on design rather than on remembering something learned previously. The problem itself is interesting because finding convex hulls has a number of applications, and there are a variety of algorithms that vary in time complexity. On the one hand, many standard algorithm design techniques can be applied to generate the algorithms, but on the other, convex hulls can also be found by relying on visual intuition. This allows us to watch the interplay between people solving a problem themselves and trying to design a computationally efficient algorithm. The use of geometry as a domain does have a potential disadvantage. People's visual intuition is not well understood, and it may lead to design strategies quite different from those that are easy to automate.

2.5. The subjects and the protocols

The first subject (S2) has a Ph.D. in computer science and is moderately sophisticated in theory and algorithm design, but knows little about convex hulls or complexity theory. The experiment was conducted informally in S2's office with a tape recording made of the proceedings. Before the experiment, we informed S2 that we were studying algorithm design, but did not suggest that any particular approach was of interest. The problem was specified informally, and S2 was asked to "think out loud" while working on the algorithm. While working, S2 made a number of diagrams on the blackboard which were copied by the experimenters. Occasional questions were asked of the subject during the analysis. In the first fifteen minutes of the session, S2 developed an algorithm to find the convex hull of a set of points, based on generate and test. In the remainder (about an hour's worth) S2 developed a second, more complex algorithm based on the divide and conquer paradigm.

The second subject (S4) is a graduate student in computer science who is fairly sophisticated in algorithm design and complexity theory but knows little about convex hull algorithms. Again, a tape recording was made of the proceedings, but this time the subject drew a number of diagrams on paper. In this experiment, it was suggested that S4 try a divide and conquer approach to the convex hull problem. Within about fifteen minutes, S4 sketched a solution. At the prompting of the experimenter, S4 spent the next fifteen minutes writing down a description of the algorithm, filling in a few more details, and describing the time complexity in more detail.

3. Case Studies

Before considering the lessons learned from the protocols about algorithm design, it is useful to have an overview of the problem and of the behavior of the subjects. The reader may wish to skim these examples now, then return to them later.

3.1. The problem

The problem is to design an algorithm to find the convex hull for any given set of points. The convex hull is the smallest subset (in the sense of set inclusion) of the points that, when connected in a convex polygon, contains all the other points. Figure 3-1 shows an example of a point set and its convex hull. The solution can be either the set of points on the hull given in arbitrary order, the points listed in the order they would appear on the polygon, or the polygon described by line segments. The problem description given to the subjects was ambiguous about what was being asked for, but the subjects generated polygons as solutions.

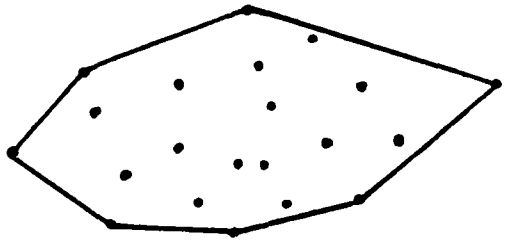


Figure 3-1: A point set and its convex hull

3.2. Overview of behavior of S2 on Algorithm GT (generate and test)

An overview of S2's behavior can be obtained in part by reading the edited protocol given in Figure 3-2. To save space here, many interruptions, side comments, and false paths have been omitted, although they were important in the analysis. Each line has a label, such as L77, numbered according to the original protocol. In the figure, S2's behavior has been segmented into a series of short sections, called *episodes*, each of which contains essentially a single event or topic.

Episode 1
L1 E: [>>Minute1<<] Do you know what a convex hull is?
L2 Vaguely. Why don't you give me all the definitions.
L4 E: Suppose you have a set of points, ok...
L7 E: well there's several ways you can define it.
L8 E: Either the polygon that encloses all the other points
L9 E: or the set of points on the polygon.
L10 Yes, that's a hard problem
L11 and I don't know any of the algorithms.

Episode 2
L18 Right. The problem is you've got a bunch of points.
L19 Let's not worry about how they're specified.
L20 What's a reasonable solution?
L21 [>>Minute 2<<] Let's start with some point.
L25 Either a point is on the convex hull or its not, right?
L27 And the question is how to make this decision.

Episode 3
Episode 3.1
L28 Let's take a few points here. (Draws 4 points.)
L29 Well, that's not a good example,
L30 because all four of them are on the convex hull.
[S draws figure with 5 points not all in hull.]
L35 OK, let's suppose I start with a point here.
L36 And I'll just draw a line to some other point, right.
L42 Now I can go in any one of three directions from this point.
L43 [>>Minute 3<<] I conjecture that
L44 if it's the case that I can choose two points,
L45 such that I can go on either side of the given line,
L46 then this line can't be on the convex hull.
L47 And I had better retreat.
L51 Here's A, B, C, D, E, right [labels points in Figure 1].
L52 So I can go from A to B.
L53 And I find that from B
L54 I can go either to C or D,
L55 and C and D are on different sides of this line...
L61 then clearly this line can't be on the convex hull.
Episode 3.2
L63 Let's retreat, uh, [>>Minute 4<<]
L64 back... back to A
L65 and choose some other point.
L66 And this time we'll chose C.
L67 Right? So now I have a line from A to C.
Episode 3.3
L68 Let me rephrase the problem to make it even harder on myself.
L70 I had a line drawn from A to B, OK.
L71 And I'm considering these five points here.
L72 What I want to do is rephrase the problem
L73 so that I'm starting at point B. [pause]
L74 E: Why point B?
L75 Because if I start at point B
L76 and I go to A
L77 then either route here,
L78 I still have that problem
L79 and I want to retreat
L80 back to point B.
L81 But it turns out that no matter what...
L82 which direction I go in from point B
L83 I'm going to have the same problem. [>>Minute 5<<]
L84 So point B can't possibly be on the convex hull.
Episode 3.4
L86 So let's go back once again to starting at A,
L87 because A is going to be on the convex hull.
L88 And we don't want to retreat too far, right?
L90 OK, so I'm going to retreat here
L91 from B back to A
L92 and go to point C instead.
L113 [>>Minute 6<<] And I see that, um,
L114 all the points are to one side of the line AC.
L115 So I've got a candidate.
L116 Now I'm at C and now I'll go again.
L117 Choose some other point.
L118 Suppose I choose B. [pause]
L119 A goes to C goes to B.
L120 Um, now I see that uh, [pause]
L121 there are points on either side of the line C-B, right,
L122 there's E and there's A.
L123 [>>Minute 7<<] I guess I have to look at A
L124 even though I've already got a line segment from it.
L125 So I know that the line C-B can't be on the hull.
L126 So I have to retreat back to C.
L128 It looks like I'm not going to come up with a linear algorithm to do this.
L129 So therefore I'll go from C to some other point.
L130 Same problem with D, so I'll go to E.
L131 And I win:
L132 all points are always on one side of the line.
L133 If fact they're always on the... the right side,
L135 if I give this directionality.
L136 If I go from E to B I lose
L137 because C is over here and B is over here.
L138 So that leaves me D
L139 and then I go back.
L140 So I've got a convex hull.
Episode 3.5
L141 E: Now could you describe your algorithm?
L142 Well, I'm not sure it's an algorithm yet, right?
L143 Because if I start at a losing point,
L144 if I were to have started at this middle point, B,
L146 [>>Minute 8<<] I would have found that...
L147 there's no point that I...
L148 I couldn't have gotten to.
L149 None of the segments from B,
L150 B-A, B-C, B-D, or BE, uh,
L151 would have given me a satisfactory line.
L152 So I would have given up on B
L153 and tried some other...
L154 some other point to start with.
L155 So, I keep doing that
L156 till I get a point that satisfies it.
L157 There must be such a point,
L158 since there is a convex hull, presumably.
L160 That sounds like the algorithm.
Episode 3.6
L168 Choose a point p0, ok.
L169 Then, uh, choose a point p1 from the remaining set of points,
L170 Draw that line segment.
L171 Um, if it's the case that
L172 there are points on both sides of that line segment,
L173 then, uh, give up on p1 and try some other point.
L174 [Pause] All right, um, and keep doing that,
L175 [>>Minute 9<<] and if you exhaust all the points,
L177 then p0 can't be on the convex hull.
L179 so you go try another point to start with.

Episode 4
L181 This looks to me like an N^2... [>>Minute 10<<]
L182 No, it's worse than that.
L228 [>>Minute 11<<] I guess it's N^3. [pause]
L231 Because, why is that?
L232 I choose a line segment.
L233 If the line segment is successful,
L234 I look at the endpoints.
L235 If it fails,
L237 I can look at up to N-1 line segments,
L239 from a given point
L243 Looking a line segment requires time N,
L244 proportional to the number of points.
L245 [>>Minute 12<<] OK so... [long pause]
L248 And if the point fails
L249 then I go to some other point,
L250 and I know that that point is not on the convex hull,
L251 so I don't have to consider it any more.
L252 So it's N^3.
L253 I know that N^3 is an upper bound,
L254 for this particular algorithm.
L255 So it's not a very good algorithm.

Episode 5
L258 This is a first shot...
L259 This is algorithm 1.

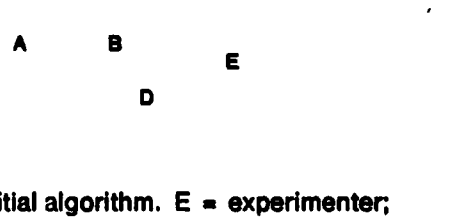


Figure 3-2: Edited protocol of S2 developing the initial algorithm. E = experimenter; unattributed lines are spoken by S2. Each line is about 2.5 seconds.

Figure 3-3 gives the major episodes in S2's design of an initial algorithm, indicating for each the name (which reflects its position in the hierarchy), the sequence of lines it covers, the number of lines, and a short descriptive phrase for the content.

E	Lines	#L	sec	Description
E1	L1-L15	15	37.5	Acquire problem
E2	L16-L27	12	30	Design generate-and-test schema
E2.1	L22-L24	3	7.5	Interrupt (E2): specification of points
E3	L28-L179	152	380	Develop algorithm
E3.1	L28-L61	34	85	Find test
E3.1.1	L28-L32	5	12.5	Get example figure
E3.2	L62-L67	6	15	Decide how to handle test failure
E3.3	L68-L85	18	45	Decide how to handle interior start point
E3.3.1	L68-L73	6	15	Detect problem
E3.3.2	L74-L85	12	30	Find: can discard interior point
E3.4	L86-L140	55	137.5	Push algorithm all the way to find CH
E3.4.1	L86-L92	7	17.5	Return to previous state after E3.2
E3.4.2	L93-L108	16	40	Interrupt (S): Segment excluded, not point
E3.4.3	L127-L128	2	5	Interrupt (S): Greater than linear
E3.5	L141-L160	20	50	Develop initialization
E3.6	L161-L179	19	47.5	Recap algorithm
E4	L180-L255	76	190	Analyze complexity
E4.1	L180-L187	8	20	Time complexity is worse than N^2
E4.2	L188-L204	17	42.5	Time complexity is N^2
E4.3	L205-L228	24	60	Time complexity is N^3
E4.4	L229-L255	27	67.5	Confirm N^3. Algorithm not good
E5	L256-L261	6	15	Termination. Algorithm is first try.
		261	652.5	

Figure 3-3: Selected episodes in S2's Algorithm GT (times are approximate).

3.2.1. Summary of algorithm

Before trying to understand the course of the problem solving revealed in Figures 3-2 and 3-3, the reader needs to have a general idea of the algorithm that S2 finally developed. This is shown schematically in Figure 3-4.

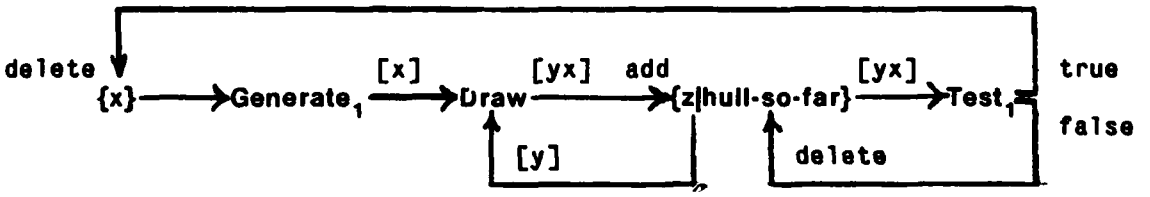


Figure 3-4: Final Algorithm GT by S2.

The algorithm starts with the original set of points, {x}, enumerates (**Generate**$_1$) them, and tests (**Test**$_1$) each one to see if it is on the convex hull. The partial hull-so-far is held in **{z|hull-so-far}**, which is used as input to the test. Actually, the partial hull is extended at each step by the operation (**Draw**) of drawing a line from the prior final point (y) in {z} to the new one (x), and the test is whether this new line could be on the hull. Thus, if the test fails (the false branch of **Test**$_1$), then the latest

point must be deleted from {z}. On the other hand, if the test succeeds, then the point can be removed from the input set {x}. Not represented in the schematic figure is S2's method for finding the initial point on the hull. This involves discarding any point that does not have at least one attached line that satisfies $\mathbf{Test}_1$ (is on the hull).

3.2.2. The story of S2's solving attempt

The solution attempt starts (Episode E1) with the experimenters giving S2 the problem. The subject immediately (E2) develops a schema for generating the points in the set and testing whether they are on the convex hull ({x}--->$\mathbf{Generate}_1$--->$\mathbf{Test}_1$--->{z}). During this episode, S2 decides not to worry about how to represent the points. The next episode (E3) is devoted to refining the schema just created. In the first subepisode (E3.1), S2 discovers a test ($\mathbf{Test}_1$) for a point being on the convex hull. This discovery involves creating on the blackboard an example figure (E3.1.1 is devoted to getting this figure, which starts with four points and then is enriched to five). The net result of E3.1 is a test for an *edge*, with a shift from points to line segments between pairs of points (moving {z} to be the input of $\mathbf{Test}_1$ and adding the operator **Draw** to draw the line). The rest of E3 is driven by S2 attempting to push the special case through the partially developed algorithm, a method we call *test-case execution*. This attempt leads to resolving several problems, with the consequent elaboration of the algorithm. E3.2 settles what to do when the test fails, and E3.3 settles what to do if the initial point is not on the convex hull. There is a special problem with initialization, since the test requires both an old and new point. With these issues cleared up, the algorithm successfully handles all the points in the example task (E3.4). This leads to recasting the initialization (E3.5) and then summarizing the entire algorithm (E.3.6). At this point the algorithm is complete. However, the subject is concerned about the time complexity of the algorithm. After several tries (E4.2, E4.3 and E4.4), S2 determines that the algorithm is N^3. S2 concludes that the algorithm is not very good and (E5) terminates the initial attempt by declaring that it is simply a "first shot".

3.3. Overview of behavior of S2 on Algorithm DC (divide and conquer)

Following the conclusion that an N^3 algorithm was not good enough, S2 produced a second algorithm. Figure 3-5 gives the major episodes in this design session. The algorithm was based on divide and conquer, for which S2 has available a moderately developed schema. In this attempt, S2 runs into more difficulties than in the first design, but eventually outlines a more efficient algorithm. S2 first decides (E2) to divide by drawing a line through the middle of the set of points, but then cannot figure out how to do the merge (E4). Later (E5), S2 decides that it might be easier to merge if the dividing line goes through one of the points so that one point will be on both hulls. This allows a merge process to work outward from the shared point (E6). S2 then decides (E6.9) how to remove the

center point, which is not on the merged hull, and how to continue the merging process in general (E6.10) by considering a limited number of cases. In E6.11-E6.13, S2 wraps up the design of the merge step by deciding when the merge can be halted, and gives a rough analysis of the time complexity of the merge. Next, S2 goes back to the divide and considers base cases (E7). In the process of testing examples, S2 discovers some degenerate cases in which the divide leaves all points on one side and does not reduce the problem at all. To get around this problem, S2 considers dividing through two points rather than one, which also slightly simplifies the merge step (E8, E9). In E10, S2 tries to take advantage of dividing segments that are on the hull, but this is a dead end since divide and conquer should build up the solutions in the merge, not the divide. It also does not get to the root of S2's problem, which is that the dividing line needs to partition the points into sets with equal numbers of elements. After a hint from the experimenters, S2 realizes this (E11) and finishes the divide by deciding to sort the points in a prepass (E11.2) and then use the midpoints. S2 takes a slight side path here in considering using a lexicographic sort and also decides to go back to the single point divide (E11.3). Finally, S2 is convinced that the algorithm works and is an N log N algorithm (E12).

E	Lines	#L	Description
E1	L1-L6	6	Decide to try divide and conquer, recall schema
E1.3	L4-L6	3	Get example in geometry-space
E2	L7-L81	75	Refine divide step
E2.4	L16-22	7	Zero-point divide (extra effort)
E3	L82-L84	3	Assume solve step can be done
E4	L85-L103	19	Refine merge
E4.3	L93-L99	7	Restrict attention to points on hulls
E5	L104-L129	26	Revise divide to line through point
E6	L130-L586	457	Resume refine merge
E6.1	L130-L150	21	Overview merge (work outward from line, stop when come in)
E6.2	L151-L158	8	Set up test case (for working outward)
E6.3	L159-L162	4	Note solve step handled by induction
E6.4	L163-L182	20	Continue refining merge (with example)
E6.5	L183-L219	37	Refine merge (first segment)
E6.6	L220-L223	4	Refine stopping condition
E6.7	L224-L233	10	Refine how to go in other direction
E6.8	L234-L253	20	Consider complexity of merge (linear)
E6.9	L254-L324	71	Decide how to handle first point in algorithm space
E6.10	L325-L410	86	Refine how to go again
E6.11	L411-L474	64	Detail stopping condition
E6.12	L475-L514	40	Case analysis of how to go again
E6.13	L515-L548	34	Consider running time of algorithm
E7	L587-L617	31	Consider divide base case
E8	L618-L636	18	Two-point divide (extra effort)
E9	L636-L669	34	Revise merge to match divide
E10	L670-L1029	360	Degenerate cases of divide (extra effort)
E11	L1030-1231	202	Finish refining divide
E11.1	L1030-1044	13	Read hint from book
E11.2	L1045-L1109	65	Want equal-size problem, solve by sorting
E11.3	L1110-L1201	92	Lexicographic sort (extra effort)
E12	L1232-L1369	138	Evaluate algorithm

Figure 3-5: Selected episodes in S2's Algorithm DC

3.4. Overview of behavior of S4 on Algorithm DC (divide and conquer)

The major episodes in S4's solution are given in Figure 3-6. Without much hesitation, S4 accepts (E2) the experimenter's suggestion to try a divide and conquer algorithm. S4 then proposes to divide (E3) by taking the median of the projection of the points on the X-axis. This idea is formed quickly, based on previous experience with geometric divide and conquer problems. Hulls are then constructed for each half, recursively (E4). However, S4 has some trouble with the merge, rejecting as too expensive the solution of finding additional edges for the merged hulls by generating and testing edges between pairs of points from different hulls (E5.8). In the process of drawing the additional edges, S4 notices that it would also be possible to divide through a shared point, and that this would allow a merge based on repeatedly eliminating concave angles starting at the median (E6.1). Whenever there is a concave angle, the two endpoints of the angle are joined and the old edges are removed; if this creates any new concave angles, the process is repeated. The merge is therefore at worst linear in the number of points on the subhulls. This key insight into the solution happens quite quickly. After determining the basic idea, S4 describes in more detail how to find the shared point (E7), tests the new merge (E8), and is then satisfied with the algorithm (E9). At the request of the experimenter, S4 summarizes the algorithm, in the process noting that there are separate cases for the division of even and odd numbers of points, and evaluating the cost of the merge (E10). The experimenter requests that S4 write down the algorithm. S4 then considers (E11.2) base cases and degenerate cases such as collinear points (E11.3), and then writes out a program sketch. In a discussion following the design session (E12-E13), the experimenter asks S4 about the complexity of the algorithm. S4 states that it is N log N, then decides that this is true only if a prepass is added to sort the points (so that the median finding can be done in constant time). S4 later noted that there is a linear median finding algorithm, which permits the algorithm to be N log N without the prepass. However, S4's first plan is to find the median by sorting on each recursive call, which would give an N $\log^2$ N algorithm, but also suggests a prepass.

4. A Model of Algorithm Design

4.1. An overview of the model

The behavior of both subjects on all three algorithm attempts presents an entirely consistent picture that fits well with the theoretical framework we have adapted and agrees with what has been reported in the literature about problem solving and design. Design, whether of programs, algorithms, or almost anything else, appears to involve hypothesizing a general schema or key idea or

E	Lines	#L	Description
E1	L1-L28	28	Understand problem
E2	L29-L33	5	Decide to try divide and conquer
E3	L34-L38	5	Refine divide (median of projection)
E4	L39	1	Assume solve step can be done
E5	L40-L98	59	Refine merge (first attempt)
E5.1	L40-L45	6	Attempt straightforward union (fail)
E5.3	L50-L59	10	Try an example
E5.5	L62-L64	3	Compare final to subhull solutions
E5.8	L69-L76	8	Get missing edges from bipartite graph
E5.10	L86-L98	13	Try to avoid generating extra edges
E6	L99-L115	17	Refine merge (successful attempt)
E6.1	L99-L106	8	Idea for revision (share point)
E6.2	L107-L112	6	Refine merge (replace concave angles)
E7	L116-L131	16	Revise divide to share median
E8	L132-L146	15	Test new merge
E9	L147-L150	4	Evaluate algorithm
E10	L151-L202	52	Describe algorithm
E10.2	L148-L191	144	Analyze cost of merge
E11	L203-L401	199	Write down algorithm
E11.2	L209-L230	22	Check boundary conditions and base cases
E11.3	L231-L259	29	Check degenerate cases
E12	L402-L444	43	Interruption: discuss how got algorithm ideas
E13	L445-L493	49	Discuss complexity of algorithm

Figure 3-6: Selected episodes in S4's Algorithm DC

solution plan and then proceeding by successive refinement.[3] The protocol shows that S2 fits this part of the scheme without doubt. In the design of the first algorithm, the first episode after the problem acquisition (E2) involves the creation of an initial schema of the algorithm (generate and test). This occurs very rapidly (which agrees with the other data that exists on human design). In part, the speed comes from the schema's simplicity; it is just a kernel of an idea, and everything remains to be done. The rest of the design time is spent in gradually refining this initial schema. Similarly, S4 accepts the suggestion of trying divide and conquer after a few seconds thought, and expands this kernel idea into a divide, a solve, and a merge. S4 fills in the divide part of the schema with very little trouble based on previous experience, then flounders for a while on how to do the merge. A successful schema follows almost instantly after realizing another key idea about dividing through a shared point. Given the paucity of data about human design, it is important to have this confirmation of the refinement paradigm.

However, successive refinement is not the whole story. The detailed construction of the algorithm is accomplished by using two closely related methods:

[3] The published evidence for this is only modest. Some work on the psychology of programming supports this [3, 9], and it is almost universally adopted in the existing systems that do programming of any complexity (see examples cited previously). Counterexamples seem to occur in tasks that have been restricted enough to become strongly combinatorial in a fixed space, so there is no constructive idea to be discovered (for example, space layout problems).

- **Symbolic execution:** Execution of the program with general symbols as input; the symbols become elaborated (with assertions) along with the algorithm (new components and assertions are added) in the process of execution. Processes in the algorithm description are symbolically executed only once.
- **Test-case execution:** Execution of the program on a specific test case, with the algorithm becoming elaborated as the process proceeds. There may be many cycles of execution if generators in the algorithm produce different test-case items.

These methods are more than just code generators. They are the major device for generating consequences and exposing problems and complications. Only the concrete situation can uncover what really must be done by the algorithms, because human memory is essentially associative and must be presented with concrete retrieval cues to make contact with the relevant knowledge. These two methods serve this purpose. Furthermore, the protocols show that these methods are used after only a small amount of refinement.

The use of successive refinement and these execution methods does not imply that design is not search in a problem space. Initial kernel design ideas are sometimes wrong and have to be abandoned. Thus, the use of these methods implies only that the search space is one of schematic structures in which some operators refine partially specified structures or assertions and other operators carry out the execution methods.

Most of the subject's design behavior seems to occur within a single problem space which is a space of algorithms. However, the subjects occasionally solve subgoals and make discoveries within the geometric task-domain space, whose components are points and lines. This is not the only way it could have happened. For instance, the subjects could have (but did not) first solved the problem of finding the convex hull in a geometrical space and then transformed this knowledge into an algorithm.

Thus, we can identify four elements that comprise the core of the subjects' general approach to finding the convex hull algorithm:

1. The very rapid development of a highly schematic key or *kernel* idea.
2. The general use of *successive refinement* for further development.
3. The main methods of *symbolic execution* and *test-case execution*, which involve execution of the (partial) algorithm against an appropriate data set.
4. The use of a single main problem space for representing the algorithm with side excursions into a domain space for testing and discovery.

These four elements do not by themselves determine behavior completely. Rather, they provide a

framework within which additional search control knowledge occurs. For instance, a part of the algorithm must be selected to refine or symbolically execute. Also, there are usually many outstanding problems to solve, and their number tends to increase, since working on one problem often creates new ones. Which of these to work on must be selected. While the exact course of problem solving is determined by the applicability of operators and methods in the problem space, typically, design steps occur in the order shown below. Many of the steps described here are similar to those found by Jeffries et. al. [9] to be used in software design, although they propose a production system model with the steps explicitly controlling the problem solving process.

- Select a problem -- the problems are exposed by symbolic and test case execution. They are then considered right away or suspended.
    - o If a problem is critical (other problems depend on it), work on it right away.
    - o If the current problem is trivial, at least for the current level of detail, do not consider it any further.
    - o If a new component has been added, refine it right away to another level of detail.
    - o If a set of components has been added, start refining them in structure order, breadth-first. The structure ordering occurs as a natural consequence of the execution methods.
- Problem solve -- try to get the kernel idea or solution plan. In an expert problem solver, several possibilities will suggest themselves. Evaluate each and take the best. This may result in declaring the problem unsolvable.
    - o If a plan or similar problem can be retrieved from memory, try to use it.
    - o If retrieval fails, try to get more knowledge about the problem by experimenting in the domain space, for example using test-case execution on the partial algorithm design developed so far.
- Structure -- lay down the basic structure such as generate and test or divide and conquer or input-process-output. This effectively decomposes a problem into subproblems, or outlines the structure of a solution to a single problem. Many new problems or goals may be added as a result of structuring. Structuring occurs naturally as a result of selecting a kernel schema and then trying to execute it.
- Elaborate -- fill in the details of the structure. Use specific knowledge appropriate to the problem or task domain, or use the general technique of symbolic or test-case execution.

Verify -- if a previous pass of symbolic or test-case execution was successfully completed with no inconsistencies or missing components, consider the (partial) algorithm verified.

- o If it is not very certain the (partial) algorithm will be used in the final solution, skip the verification.
- o If the partial algorithm is likely to be used, perform another pass of symbolic or test-case execution and check for inconsistencies, missing pieces, or unproven conjectures.
- o If an assertion is made about the task domain, the statement must be verified within the task-domain space, perhaps by test-case execution of the assertion.
- o If a very careful verification is desired (once the final solution is determined or before coding), symbolically execute the program, checking that initializations, base cases, degenerate cases, and so forth, are present.

- Evaluate -- decide on the goodness of the technique. The decision is usually based on the time complexity (or space, or simplicity) relative to other alternatives or known or estimated lower bounds. Thus, complexity analysis may be a subtask of evaluation; it also may be combined with verification.

The literature on problem solving makes a distinction between expert and naive problem solving styles. Although our subjects are hardly naive about algorithm design and theoretical computer science, there is still substantial variation in the three attempts. In the initial attempt, S2 finds the task relatively unfamiliar. S2's second attempt demonstrates both more familiarity with the convex hull problem, built up during the first try, and a moderate experience with the divide and conquer schema. Finally, S4 knows more about the design of geometric algorithms and about divide and conquer than does S2. The successive increases in knowledge are clearly apparent in the three attempts. However, we do not find a major difference in design styles among our subjects, either in the methods used or in the order in which things are done. At times, all experts are at a loss for a quick solution. For example, S2 is not immediately able to state how to divide a set of points for a divide and conquer algorithm, whereas S4 immediately recalls that sorting and taking the median is a standard technique. When knowledge is lacking, all subjects rely on general problem solving schema such as generate and test, they search the task-domain space for usable facts, and they make more use of the methods of symbolic and test-case execution. Whenever there is more knowledge, as in S4's more complete schema for divide and conquer, the extra knowledge is brought to bear and improves the problem solving, but does not affect the style.

4.2. A data-flow problem space

Looking in detail at protocols has allowed us be very specific about how knowledge, operators, and control can be represented for algorithm design. These can only be summarized here. For a few more details, see [11].

A problem solver does not know the solution to a problem, but rather must perform a variety of operations to acquire additional increments of knowledge about its nature. Thus, the partial knowledge must be encodable in some internal representation so the problem solver can retain it while taking additional steps to elaborate or extend it. What representation of knowledge is appropriate for algorithm design?

The subjects appear to work in a *data-flow problem space* (DFS) whose states represent partially specified algorithms. Figure 3-4 provides an example, although it leaves out some details in the interest of clarity. Each state is a *data-flow configuration* that includes pieces of algorithms, the data-flow links between them, the objects being manipulated, and assertions about the algorithm. The algorithm steps are represented by *process components*. There are a small number of generic process components (flow controllers such as *generate* or *test*) and some general constructs such as *apply* that can be specialized to domain operations such as draw-line-segment. The inputs and outputs of the process components are represented by *ports* connected by *links*. Process components can be further specified by *assertions*. The components and assertions together modify and control the flow of *items* that represent data objects such as points and line segments. Assertions can be attached to an object, to a process component or link, or to the space as a whole.

The small number of components allows the decision of which component to use to be based on relatively small amounts of knowledge. A more discriminative vocabulary, such as might be found in expert design, can be built up from the simpler vocabulary and can coexist with it. For example, the problem space can include a schema for and assertions about divide and conquer algorithms or about dynamic programming algorithms. DFS appears to be more appropriate than a purely algebraic, procedural representation for designing algorithms, especially before the problem is well understood, because it allows a spectrum of specification from assertional to algorithmic. We conjecture that most people's default design space is a variant of DFS and that DFS will be an interesting default space for automated design. Data-flow representations are not new; they have been studied in computer architecture research, and they have been used to describe artificial intelligence programs [19] and recently to express algorithm transformations [26]. The structure of

DFS is similar to the data-flow model first used in [17] and [19].[4]

DFS operators refine the component structures in an algorithm description and implement problem-solving techniques such as symbolic execution. Many of these operators "edit" the DFS configuration to accomplish very simple but very general tasks. For example, the operators add, delete, move, or modify process components, links, items, and assertions. They also execute components and control the problem solving process.

Before applying a problem-space operator, some mapping must be constructed between the operator's representation of entities and their representation in the current state. The more flexible and extensive this mapping, the wider the range of entities it will successfully apply to. Since the subjects' editing operators are very general, and hence flexible, the specificity in what they accomplish comes from this mapping. In DFS, a good deal of search (and/or knowledge) is required to decide how to instantiate an operator or fulfill its preconditions. For frequently used generic components, a wealth of specific search control and instantiation knowledge is available. Operator selection rules describe which process components to add to the algorithm description, how to link them to the other components, defaults for how to instantiate them, defaults for which components or parts of components to refine first, and so on. To apply operators in less common circumstances, more general techniques are needed. For example, S2 and S4 seem to use means-ends analysis to control the processes of adapting an operator to a specific problem state.

The search control rules provide knowledge to the problem solver in operational form rather than as data to be interpreted. The set of search control rules includes a variety of methods such as means-ends analysis, symbolic execution (and its specializations to test-case execution and analysis), refinement, and divide and conquer. The methods are implemented by search control rules about when to apply the method, how to apply the method, and the defaults in the method. Other search control rules determine which operators are selected, evaluate the states, and decide whether to go forward in the current space, back up, or work in another problem space.

Refinement of new components is the general process that drives the problem solving in algorithm design problems. But in the absence of any problem-specific knowledge, means-ends analysis is the default method. Means-ends analysis is the continual comparison of the current state with the desired state (or its description); the result of the comparison (a difference or an opportunity) is used to select the next operator (to reduce the difference or exploit the opportunity).

---

[4] Many of our detailed descriptions of DFS are modelled after [12].

Once at least one process component is present, symbolic execution can be used to get more information. Symbolic execution means running the process components on a partially specified computational state. The detection of difficulties and their solution leads to continually refining the process and computational state. If this is not sufficient, execution in the task environment can be tried. Test-case execution is a variant of symbolic execution in which all items in a set are examined individually rather than being represented by a single symbolic item. Test-case execution is more time consuming than symbolic execution because it is linear in the number of individual items for which the algorithm is executed whereas symbolic execution is linear in the size of the algorithm structure. Thus, while the subjects use test-case execution if needed to make progress, they usually shift back to symbolic execution when they can generalize a step sufficiently. Symbolic execution is used in a number of different circumstances:

- To elaborate the algorithm description (often the test-case variant is used for this).
- To verify an algorithm (if no difficulties are identified).
- To describe an algorithm (intermediate and low level refinements may or not be included).
- To guide problem solving in domain space (often the test-case variant is used for this).
- To analyze the time complexity of an algorithm (naive analysis is mostly a matter of counting nested generators, as illustrated by S2's difficulty in identifying a factor of N corresponding to a reset of a generator).

4.3. The task-domain space

The application-domain space is the work-horse space for the test-case execution of algorithms, a space for problem solving about the domain, and a source of key insights for designs. In the convex-hull design problem, the task-domain is a geometric space. The representations in this space are the geometric figures, partly drawn on the board, using points, line segments and polygons, and involving relations between objects such as above, between, inside, and convex. This problem space has a number of geometry-specific operators (create a point, construct a line from x to y), as well as some general operators (find, partition, test, enumerate) that are typically specialized in geometric ways (partition a point set, enumerate pairs of points). It also has a general perception operator which is specialized to the space in that it reflects spatial, geometric knowledge.

Test-case execution requires a number of operations in the domain space. First, before executing a test case of a data-flow configuration, an example or test case must be produced (for example, a small set of points). The task space is the source of this example. The subjects clearly have rules for evaluating as well as generating examples and will remark on an example being too simple. During

test-case execution, assertions such as predicates on test components (for example, "are all points on one side of the line?") may have to be evaluated in the task space. Also, in the process of test-case execution, items from the example may be produced by a generator or stored in a memory. The representation in the geometry space of memory access operations may not correspond directly to the DFS algorithm description. For example, storing a point in a memory is sometimes recorded in the geometry space by drawing a line on a figure on the blackboard from the most recently stored point to the new point. In fact, many data-flow space operators (such as draw a triangle) update the example in the geometry space. This leads to much ambiguity in the data-flow representation about whether or not intermediate results are stored.

Some problem solving occurs within the task space. Generating test cases (and attempting to find counterexamples) is one category of such problem solving. Other examples are finding (visually, not algorithmically) the convex hull of point sets and comparing (with means-ends analysis) two subhulls with the final hull to find differences. "Proofs" of conjectures occur partially within the space, usually by demonstration on an example and some argument about generalization. However, most problem solving within the task-domain space consists of just a few operator applications.

4.4. Discovery

Design and other difficult problem solving is punctuated by moments of *discovery*. These can be identified as the sudden emergence, without apparent preparation, of new knowledge which subsequently plays an important role in the solution attempt. These are the moments when something new and important is suddenly "seen." Within the total picture we have just sketched, many of the kernel ideas are discoveries, with successive refinement and symbolic execution being the working out of details. Understanding the nature of these discoveries is a central issue in obtaining a system that can discover algorithms on its own. Little work on discovery has been reported in the artificial intelligence literature. The AM and Eurisko programs [6, 15] are open-ended concept discovery and exploration systems that create interesting new concepts, but they are not problem-solving systems that focus on solving a particular problem and make discoveries relevant to that problem.

Let's look at how discoveries occur in the protocols. One example is the discovery of $Test_1$ in S2's Algorithm GT. We will treat this in some detail, for the discovery seems particularly creative and sudden from the protocol. The relevant fragment occurs at L35 to L46, which we reproduce from Figure 3-2. S2 has just started test-case execution and has drawn a sample figure of five points on the board (without labels):

L35 Ok, let's suppose I start with a point here.
L36 And I'll just draw a line to some other point, right?
L42 Now I can go in any one of three directions from this point
L43 I conjecture that
L44 if it's the case that I can choose two points,
L45 such that I can go on either side of the given line,
L46 then this line can't be on the convex hull.

In L35, S2 generates a point ("A" in Figure 3-2) and then draws a line to another point, "B" (L36). There follows an irrelevant interruption by the experimenter and a response from the subject (L37 to L41, not shown). Then the subject comments (in L42) that there are three possible directions to go from B and immediately thereafter enunciates clearly and completely $Test_1$ (L43 to L46). This seems to come out of the blue -- a genuine discovery. It is rather neat and serves S2 through two attempts at an algorithm. The test itself is not particularly obvious. To be sure, there is L42 as an antecedent, but that also calls for explanation -- what made S2 consider the three directions just at that point and what bearing, if any, does that have on the discovery. Also, we must explain why the test determines whether *line segments* are on the hull whereas the original goal was to generate and test *points*.

We can analyze this discovery from the *problem behavior graph* [21] that summarizes S2's search behavior. The relevant fragment of the problem behavior graph is shown in Figure 4-1. The nodes are the states in S2's problem space, numbered in order of occurrence. The arcs show the application of the operators in either the algorithm space DFS (labeled execute, refine, add input, add component), in the geometry space (labeled perceive, draw line), or the goals to be solved (labeled refine, instantiate). The results of applying an operator occur as the next state (the next node), and in effect move the subject through the graph. The graph is too big to fit into the figure without folding back on itself (the dotted lines). The inset shows the graph drawn whole, but highly compressed, so its structure can be appreciated. The representation of subgoals can be seen clearly here: the branch for the goal is broken in the middle with a vertical dotted line and then commences horizontally again further down the page. The entire subgraph to the right of the break is the search that occurs to achieve the subgoal. When the behavior internal to the subgoal finishes, new behavior continues along the vertical dotted line and the following horizontal line. Thus, goal behavior is represented twice, once as the search tree for the behavior inside the goal, and also as a single horizontal-vertical-horizontal branch showing the goal attempt analogously to a single operator application. Finally, underneath the branches are the protocol line numbers (for example, L35 below node [1]) corresponding roughly to what is happening at that point.

As is apparent from the branching in the figure, S2's problem-solving behavior is a search for an

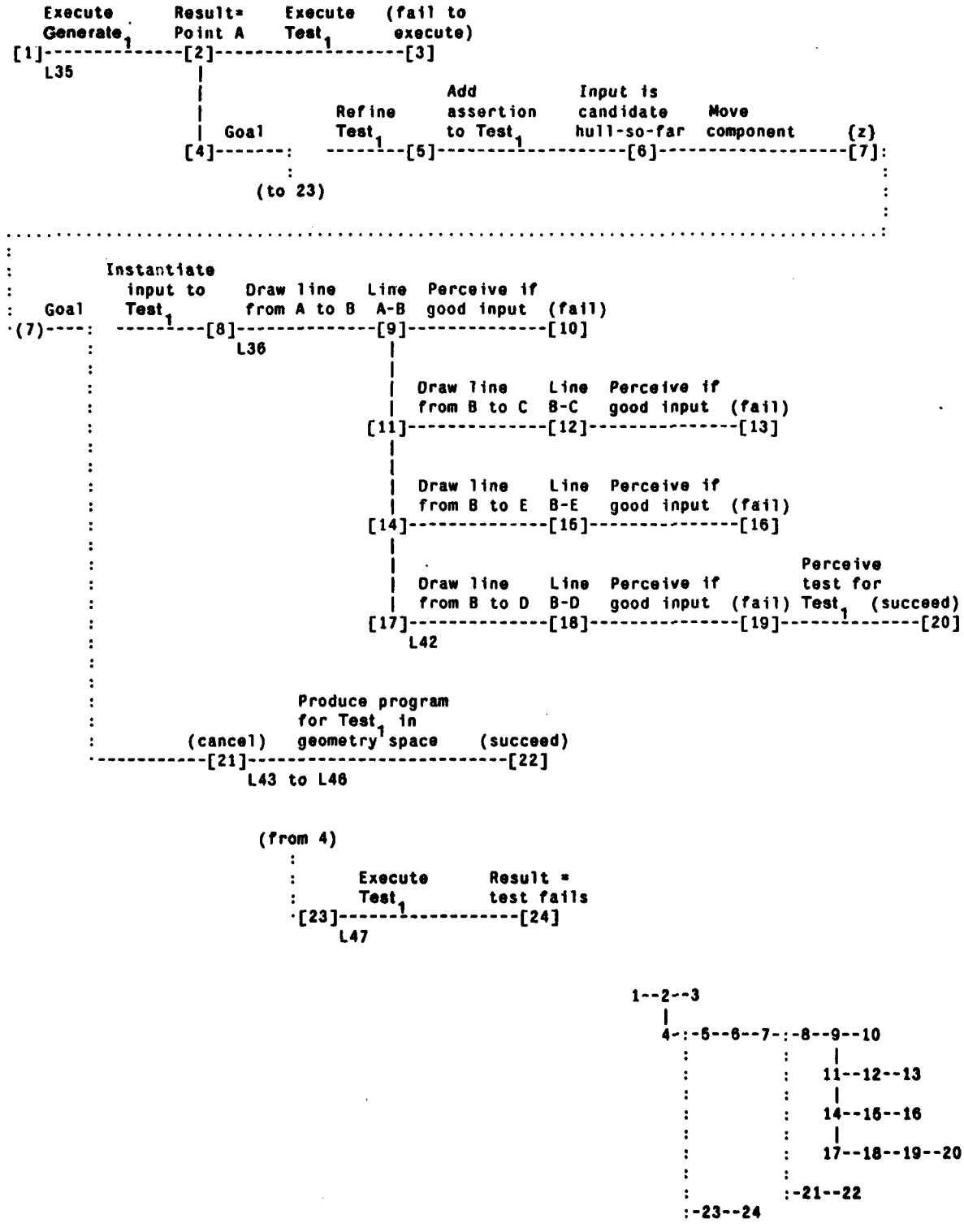


Figure 4-1: Problem Behavior Graph of S2 on fragment of Algorithm GT (simplified).

algorithm. S2 searches primarily in the algorithm space and occasionally solves subgoals in the geometry space. These are usually quite limited behaviors for executing or instantiating DFS components or assertions. However, in a few instances, such as nodes [6] to [19] here, substantial problem solving occurs in the geometry space.

To set the stage for the discovery, prior to node [1] S2 has the basic schema {x}--->Generate$_1$-->Test$_1$-->{z} and has begun test-case execution by constructing the five-point figure. Moving into the method, S2 first generates a point "A" (node [2]). That particular point is chosen because S2 knows that it is in fact on the hull.[5] S2 then moves to execute Test$_1$ on point A, but this fails because there is no actual test there (node [3]). Thus, S2 backs off and creates the subgoal of refining Test$_1$ (node [5]). This is the normal way test-case execution and symbolic execution work to extend an algorithm.

The given form of input to the test is inadequate (node [5]), because S2 sees no way to test a single point algorithmically. The test input must be a larger structure, so S2 modifies the form of the input to the test from a point to a candidate convex hull (nodes [6] and [7]). Since {z} is already the hull-so-far, this involves moving {z} from the output of Test$_1$ to its input.

S2 must now find a suitable part of the hull to use as an input to Test$_1$, in order to discover an actual test predicate. Thus, S2 sets up the subgoal of *finding this instantiation (node [7])*, which implies going to the geometry space. S2 draws a line from A to another point ("B"), tentatively incorporating the segment A-B into the hull-so-far (node [9]). Point B could be selected because it is not on the hull or because it is a nearby point; the evidence is not clear. The line could be drawn from A to B because the hull is a polygon, but it could also be drawn just as a way to keep track of an ordered set of points. But S2 still cannot see how to determine whether the segment A-B is on the hull (node [10]), so A-B is unsuitable as the required input. Therefore, S2 again prepares to draw a segment from the last point (B) to another point. By now, having failed twice, S2 considers the alternatives that have not yet been selected, which are the three points C, E, and D. All are on the hull, and S2 mentally draws a line to each possibility (nodes [11] through [19], protocol line L42), as shown in Figure 4-2.

Each triple (A-B-C, A-B-E, A-B-D) is assessed as a potential input and is found inadequate (nodes [13], [16] and [19]). However each of these assessments yields a bit of partial information, and the

---

[5] The only facts known about the points are whether or not they are on the hull (see L30), so this is the only possible selection criteria. Additional evidence that the selection of A is deliberate is found in episode E3.3 when S2 decides to try starting from a point *not* on the hull to see how the algorithm will handle a "harder" case.

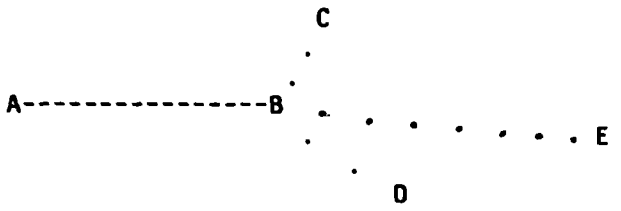


Figure 4-2: Figure for discovery of $Test_1$.

co-occurrence of all these bits of information in fact yields a successful predicate for $Test_1$ (node [20]). That is, what is perceived (discovered) is not a solution to the goal of finding a suitable instantiation, but a solution to the main problem of finding a test for being on the convex hull. Thus, S2 cancels the instantiation subgoal (node [21]) and then, on returning to the subgoal of refining $Test_1$, constructs the procedure for the test (node [23], protocol lines L43 to L46).

To see the actual discovery, consider the configuration in Figure 4-2, which S2 creates in pursuing the instantiation goal. It consists of three lines radiating from A-B, with the middle one almost an extension of A-B. With these lines in place, S2 notices that the convex hull cannot lie above the line A-B (for example if it goes from A to B to C) because then D and E would not be inside the hull. But also the hull cannot lie below A-B (for example if it goes from A to B to D) because then C and E would not be inside the hull. There are only two sides to the line A-B, so the edge A-B cannot be on the hull at all.

Although the discovery was made with three points, S2 generalizes the test to use two points in the conjecture of lines L43-46. Although not part of the discovery, it is interesting that the program for $Test_1$ is in geometry space, not in algorithm space (DFS). Furthermore, the test is executed many times during the rest of the session, but it is never recoded as an explicit set of components in algorithm space.

The recognition involved in the discovery of the test involves drawing lines not explicitly part of the algorithm, seeing the completion of polygons, and reasoning in the geometry space. In an important sense, the discovery is an accident. S2 was not trying to find $Test_1$ at that point. On the other hand, the degree of preparedness was phenomenal. There was an active supergoal to refine $Test_1$ and the test explicitly being made -- whether the three points (two segments) was adequate to determine the hull and thus be a suitable input -- was intimately related. Still, it required the fortuitous conjunction of the partial results to provide the local context in which the predicate for $Test_1$ could be recognized.

S2 makes several other discoveries during the construction of Algorithm GT which we will not analyze in detail here. For example, in the preceding discovery, S2 assumes it is sufficient to look at the set of points remaining (not already on the hull) to test whether there are points on both sides of the edge. In a later situation (L120-L124), a segment is clearly not on the hull (that is, the fact can be perceived directly in geometry space) but points from the remainder set are all on one side. S2 notices a violation of the earlier assumption. Points from the entire set, not just the remainder set, have to be considered. Here the discovery does not satisfy a previously unsatisfied goal, but does pertain to an unverified assumption about $\mathbf{Test}_1$. Another discovery occurs when S2 finds a second segment on the hull (L133-L135). Here, S2 notices that when a segment is on the hull, the points are not only always on one side, they are always on the *same* side if the segments are given directionality. This observation yields an additional assertion about $\mathbf{Test}_1$.

Both subjects make the same critical discovery, though in different ways, in their divide and conquer algorithms. They both discover that is much easier to merge the convex hulls if the points are divided so that one point is shared by the two resulting point sets (and is therefore shared by the two hulls in the subproblem solutions). Both S2 and S4 originally divide the input points into two disjoint sets by drawing a line through the middle, but then have trouble figuring out how to merge. In both cases, the discovery of the possibility of dividing through a central point is quickly followed by progress in refining the merge step.

A number of factors contribute to causing S2 to focus on including the center point in both hulls. S2 is looking at the figure Figure 4-3, with the goal of finding a way to restrict attention to the points on the two hulls that would be kept in the merged hull.

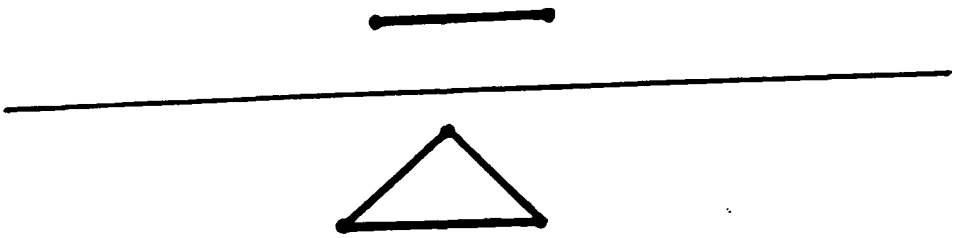


Figure 4-3: Initial division of points and solution of subproblems by S2.

First, the experimenter interrupts and asks whether S2 is assuming that all points on the hull after the merge are on one of the two hulls in the subproblem solution. The interruption seems to be misunderstood by S2 to be asking whether each point is *exclusively* in one of the two hulls, perhaps suggesting that some point should in fact be included on both hulls. Also, S2 seems to shift focus to

points *not* on the merged hull, since restricting attention to points *on* the hull does not lead anywhere. The center point is the only such point. Finally, several physical features in the diagram itself focus attention on the center point. First of all, it is the center of the picture. Also, it is at the tip of one triangle and would also be the tip of another triangle if the figure were completed.[6] As a result of all these factors, S2 considers including the center point on both hulls. This provides the opportunity to start the merge at an interesting point that is symmetrical for both subproblems.

The other subject, S4, comes to the same conclusion by a different discovery path. S4 compares the two subhulls with the desired merged hull (which is easy to construct in geometry space), notices that there are some edges missing, and considers exhaustively generating the possible missing edges by drawing lines from points on one subhull to points on the other. After drawing in some of the edges (Figure 4-4), S4 concludes that most edges are not needed and that constructing them will be too expensive.

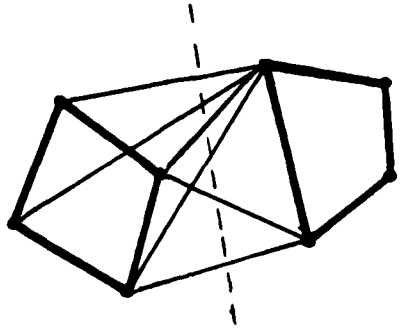


Figure 4-4: Merge attempt in a figure by S4.

S4 is focused on the new lines trying to decide which one are useful and which are not. At this point, enough lines have been drawn on paper so that two convex polygons share a point near the center of the figure. Since S4 has already mentally deleted some of the extraneous lines (the ones to "extremal" points), these connected polygons are readily apparent. S4 is familiar with other geometrical merging algorithms that start from a shared point (reported later in protocol). Therefore the visual reminder of the possibility of a non-disjoint division, after the previous difficulty in finding a merging process, gives S4 enough incentive to change the divide step to use a line *through* a point.

Thus, we see that discovery is the sudden viewing from a new perspective of a structure (or technique) that is already in existence for another purpose. Often the discovery solves a previously posed but unsatisfied problem, but sometimes it is an unlooked for refinement. Thus, we might say that the problem solver is doing the right thing for the wrong reason, and that this is made possible by

---

[6] Other evidence for the tendency of subjects to complete polygons that are only missing one side is described in [13].

the existence of a prepared mind.

The discoveries we have observed take place in the geometry space, whereas the problems they solve are posed in the algorithm space. In particular, visual noticing seems to be a combination of unsatisfied goals in the background and certain involuntary operators in the task space. At least in geometry space, there are certain configurations of data items that lead to recognition or automatic inferences (for example, completing polygons that are missing one edge and seeing polygons in regular patterns of points). Discoveries often occur when a subject is looking at a figure during test-case execution. Particularly when subjects are lost, they repeat test-case executions or stare blankly at figures.

Discoveries and refinements also occur when a subject is explaining an algorithm. Explanation is not a neutral activity; it can be dynamic problem solving, not just a repeating of past history or a simple readout of an algorithm structure. Subjects sometimes explain algorithms at the request of the experimenters, but often produce explanations for their own sake when they do not fully understand how to proceed. The process of explanation is another instance of symbolic execution and holds the same possibilities of noticing untested assumptions, untried cases, inconsistencies, and fortuitous configurations in sample figures. For example, S4 adds a prepass to the divide and conquer algorithm to sort the points during an explanation of why the algorithm was N log N, *after* the design session officially is over.

5. A Comparison of Designs

Figure 5-1 shows the number of lines of protocol corresponding to a variety of design activities for Algorithm GT and Algorithm DC of subject S2 and Algorithm DC of subject S4. As the totals show, S2's second design takes 5 times as long as the first. Why? S2 arrives at a more efficient algorithm, but is it 5 times as complex? Our analysis has shown that S2 uses essentially the same discovery and derivation techniques, applying even more extensive algorithm knowledge in Algorithm DC. Furthermore, there is no interference from the first algorithm. In fact, there is a useful carry over of the test whether a segment is on the hull. Similarly, S4 finds a divide and conquer algorithm in less than half the time it takes S2. Is this because S4 is smarter?

	Problem Acquisition	Main Design	Extra Effort	Algorithm Evaluation	Interruptn	Total
S2, Alg GT	18	114	0	53	76	261
S2, Alg DC	0	496	282	68	533	1360
S4, Alg DC	28	172	0	10	26	268

Figure 5-1: Decomposition of design activities, in protocol lines of 2.5 seconds/line for S2 and 3 seconds/line for S4

Figure 5-1 separates the time devoted to the main design (as it finally emerged) from the times for understanding the problem specification, for extra effort not contributing directly to the final algorithm, for evaluating the final design, and for interruptions not relevant to the design task. It reveals that the main design time for S2's second algorithm is 4.4 times that of the first, but that if the extra effort is considered, the ratio is 6.8. If we define the difficulty of the subject in designing the algorithm to be the ratio of time spent on extra effort to time directly relevant to the final design, then S2 has 0 difficulty with the first algorithm, but 57% difficulty with the second. S4 has 0 difficulty. How do we understand these differences?

The number of components used to represent an algorithm in DFS provides a measure of the structural complexity of the final algorithm. Though simple, it reveals the nature of the subjects' processing. Figure 5-2 further subdivides the activities into adding components, adding assertions, executing (symbolic and test case) partial algorithms, and other problem solving. (The decomposition of the extra effort is discussed below.) Each row includes the number of components and then, for each type of activity, the time per component (in protocol lines).

The time per component in the main line of the design of the two algorithms is approximately constant for the activities of adding components, adding assertions, and execution, totalling 18 protocol lines per component (45 sec) for S2 and 12.3 lines per component (37 sec) for S4. The number of assertions per component is about 1 (.86, .96, and .82 for Algorithms GT and DC of S2 and Algorithm DC of S4 respectively). In problem-space terms, this means that the great bulk of activity is not problematic, but is proportional to the structural complexity of the algorithm being designed. Thus the factor of 4.4 between S2's algorithms and the factor of 2.4 between S2's Algorithm DC and S4's Algorithm DC are simply because S2's Algorithm DC has more components.

The results for S4 are similar to those for S2 with a few exceptions. The number of protocol lines per component is quite similar for adding components and assertions, but less time is spent on symbolic execution and more on other problem solving. In fact, since S4 spends 3 seconds per line of protocol,[7] S4 actually spends a little more time than S2 in adding components and assertions. These differences can be explained by the fact that S4 has a broader base of algorithm design experience than S2 and spends more time in expert design activities -- evaluating general principles and considering analogous algorithms -- and less time in the naive design activity of test-case execution.

The analysis reveals that S2's extra effort in Algorithm DC is devoted to four distinct problems. The

---

[7] S4 pauses more frequently and for longer periods of time than S2.

	Number of Components	Adding Components	Adding Assertions	Symbolic and Test-case Execution	Other Problem Solving
S2, Algorithm GT					
Main line	7	1.9	4.3	10.1	0.0
S2, Algorithm DC					
Main line	27	1.8	5.0	10.5	1.0
Extra Effort					
0-point divide (Episode E2.4)	2	2.0	0.5	0.5	0.0
2-point divide (Episode E8)	3	2.3	2.3	8.3	5.3
lexicographic sort (Episode E11.3)	2	1.0	4.5	18.0	0.0
lost on divide (degenerate cases) (Episode E10)	3	1.7	2.0	42.0	12.3
S4, Algorithm DC					
Main line	14	1.9	4.4	4.0	2.0

Figure 5-2: Breakdown of main design and extra effort activities in protocol lines per component

first three are additional algorithm construction steps similar to the main design but superseded by subsequent search (for example, since the dividing line finally passes through 1 point, the 0- and 2-point solutions become extra effort). This extra effort fits into a standard search framework as additional branches.

The fourth case is different. It is much longer (62% of all extra effort), and the construction of three extra components, counted at the standard 18 lines per component, accounts for only 31% of the work. S2 becomes lost in this problem, worrying about how to take advantage of degenerate cases rather than how to avoid them, and wanders around, repeatedly failing to make progress and not retrieving enough knowledge to posit new components and investigate them.

This fourth case makes evident the role of symbolic and test-case execution. S2's behavior shows that the algorithm structure is not grown in a simple depth-first fashion, but rather that the execution steps scan this partial structure to find the next place to extend it. Thus, in the last extra-effort case, the amount of time devoted to execution steps is very large (42 lines per new component), compared with that for the main line (10 lines per component), because the subject cannot find any new information to use, so repeatedly scans over the structure in vain.

6. Discussion

Our analysis of the human design process suggests that an automated design system could be built around the same problem-space model that people seem to fit (one benefit of studying human design). We have applied the operators we postulate for the data-flow space and geometry space to

explain most of the major jumps and shifts in the protocols (although some steps remain to be explained in detail), which gives evidence that the problem-space model accounts for our subjects' behavior. From this behavior of S2 and S4, we conclude that an automatic system of analogous design should have the following properties:

- Algorithm representations must be deliberately ambiguous in order to handle partial states of knowledge and assertions that have not been fully integrated during the design process.
- A variety of algorithm design schema such as generate and test or divide and conquer must also be available.
- A variety of search control methods must be available. Symbolic and test-case execution and means-ends analysis are versatile general methods.
- To provide for flexibility or robustness in unforeseen situations (which are by definition a common occurrence in design), a few general purpose operators with a powerful mapping process (such as means-ends analysis) should be used. Local adaptation to bridge the gap to the best available knowledge is preferable to providing many detailed operators whose applicability is determined by a simple pattern match.

We have also made some observations about the process of human discovery that could be incorporated into an automatic design system to allow it to exhibit a good deal of flexibility, robustness, and creativity:

- Discovery involves a prepared problem state (unsatisfied goals in the algorithm space).
- Discovery requires an act of recognition in the task-domain space.
- As a result, discovery means doing the right thing for the wrong reason.

Since we have looked only at a small number of protocols and subjects, many interesting questions remain unexplored. How does the design process vary from person to person? How much does it vary with the domain of the algorithm being constructed? Is the main design time always proportional to the number of components in the design? How important is the process of analogy for adapting known algorithms to solve new problems? We expect that the study of additional human protocols will shed some light on these questions and may also give us some hints about how to automate the process of learning. For example, we would like to know whether the use of analogy is a component of learning, and whether multiple spaces are important for automated design. The main spaces in the designs studies so far seem to be only the task space and DFS, with the task-domain space used as a subspace of the main space DFS. If multiple spaces turn out to be important, we will have to decide how to *build* different problem spaces to work in and how to construct operators. Little research has been done on this. It may turn out to be some sort instantiation process of specializing more general spaces to particular situations. It may shed some light on how algorithm design is learned.

We are not currently aware of any serious limitations in the basic data-flow and successive refinement approaches at this time, but we have hardly resolved all the issues in building an expert design system. One issue is how to store and access the large volume of knowledge that will be required for high performance; this also implies that good search controls will be necessary. Another issue is how the subjects (and hence automatic design systems) switch between problems spaces. For example, how does a problem solver build a program in geometry space or algorithm space from remembered previous problem-solving actions? Finally, we still need to explore in more detail the process of a visual "noticing" and the details of the geometry space. This area is poorly understood despite an immense amount of research on visual perception.

We are actively studying the issues involved in building an expert design system and have begun an implementation of a simulation system that will recreate the algorithms designed by S2 and S4. We will also continue studying human protocols. As we add more and more algorithm and search control knowledge based on these studies, the system will gradually be extended into an automatic algorithm discovery system.

Acknowledgements

We thank our subjects for their time and interest in this project. David Steier carefully read a draft of this paper and provided many helpful comments. Steier, Edward Pervin, and Brigham Bell are helping to implement a simulation system to test our hypotheses.

References

1. Balzer, R. "Transformational implementation: an example." *IEEE Transactions on Software Engineering SE-7*, 1 (January 1981).

2. Bibel, W. and Horning, K. M. LOPS - A System Based on a Strategical Approach to Program Synthesis. Proccedings of the International Workshop on Program Construction, France, September, 1980.

3. Brooks, R. *A model of human cognitive behavior in writing code for computer programs.* Ph.D. Th., Carnegie-Mellon University, 1975.

4. Cheatham, T. E., Townley, J. A., and Holloway, G. H. A system for program refinement. Proceedings of the 4th International Conference on Software Engineering, September, 1979, pp. 53-63.

5. Darlington, J. "A synthesis of several sorting algorithms." *Acta Informatica 11*, 1 (1978).

6. Davis, R. and Lenat, D. B.. *Knowledge-based Systems in Artificial Intelligence.* McGraw-Hill, 1981.

7. Ericsson, K.A. and Simon, H. A. "Verbal Reports as Data." *Psychological Review 87*, 3 (May 1980), 215-251.

8. Green, C. C., Gabriel, R., Kant, E., Kedzierski, B., McCune, B., Phillips, J., Tappel, S., and Westfold, S. Results in knowledge-based program synthesis. Proceedings of the Sixth International Joint Conference on Artifical Intelligence, Tokyo, Japan, August, 1979, pp. 342-344.

9. Jeffries, R., Turner, A. A., and Polson, P. G. The Processes Involved in Designing Software. In *Cognitive Skills and Their Acquisition*, John R. Anderson, Ed.,Lawrence Erlbaum Associates, 1981, ch. 8.

10. Kant, E.. *Efficiency in Program Synthesis.* UMI Research Press, 1981.

11. Kant, E. and Newell, A. Naive algorithm design techniques: a case study. Proceedings of the European Conference on Artificial Intelligence, Orsay, France, July, 1982.

12. Laird, J., and Newell, A. Planning: A problem-space perspective. (in preparation)

13. Larkin, J. Spatial Reasoning in Solving Physics Problems. Tech. Rept. C.I.P. # 434, Carnegie-Mellon University, Department of Psychology, 1982.

14. Lee, S., De Roever, W. P, and Gerhart, S. L. The evolution of list-copying algorithms and the need for structured program verification. Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, January, 1979, pp. 53-67.

15. Lenat, D. B. Heuretics: Theoretical and Experimental Study of Heuristics Rules. Proceedings of the National Conference on Artificial Intelligence, August 18-20, 1982, pp. 159-163.

16. Low, J. R. "Automatic data structure selection: an example and overview." *Comm. ACM 21*, 5 (May 1978).

17. Moore, J. A. *The Design and Evaluation of a Knowledge Net for MERLIN*. Ph.D. Th., Carnegie-Mellon University, 1971.

18. Morgenstern, M. Automating the software design process for management information systems. IEEE Computer Software and Applications Conference, November, 1977, pp. 642-647.

19. Newell, A. Heuristic programming: Ill structured problems. In *Progress in Operations Research*, Aronofsky, J., Ed.,Wiley, 1969, pp. 360-414.

20. Newell, A. Reasoning, Problem Solving, and Decision Processes: The Problem Space as a Fundamental Category. In *Attention and Performance VIII*, Nickerson, R., Ed.,Erlbaum, Hillsdale, N.J., 1980.

21. Newell, A. and Simon, H.. *Human Problem Solving*. Prentice-Hall, 1972.

22. Petry, F. E., and Biermann, A. W. Reconstruction of algorithms from memeory snapshots of their execution. Proceedings of the 1976 Annual Conference, ACM, New York, 1976, pp. 530-534.

23. Rich, C. *Inspection Methods in Programming*. Ph.D. Th., Massachusetts Institute of Technology, June 1980.

24. Rovner, P. D. Automatic representaton selection for associative data structures. Tech. Rept. TR10, The University of Rochester, Computer Science Department, September, 1976.

25. Shaw, D., Swartout, W., and Green, C. Inferring LISP programs from examples. IJCAI 4, Tbilisi, USSR, 1975, pp. 260-267.

26. Tappel, S. Some Algorithm Design Methods. Proceedings of the First Annual National Conference on Artificial Intelligence, August 18-21, 1980, pp. 64-67.