EURISKO: A Program That Learns New Heuristics and Domain Concepts

The Nature of Heuristics III: Program Design and Results

Douglas B. Lenat
Computer Science Department, Stanford University, Stanford, CA 94305, U.S.A.

ABSTRACT

The AM program, an early attempt to mechanize learning by discovery, has recently been expanded and extended to several other task domains. AM's ultimate failure apparently was due to its inability to discover new, powerful, domain-specific heuristics for the various new fields it uncovered. At that time, it seemed straight-forward to simply add 'Heuristics' as one more field in which to let AM explore, observe, define, and develop. That task—learning new heuristics by discovery—turned out to be much more difficult than was realized initially, and we have just now achieved some successes at it. Along the way, it became clearer why AM had succeeded in the first place, and why it was so difficult to use the same paradigm to discover new heuristics. In essence, AM was an automatic programming system, whose primitive actions were modifications to pieces of LISP code, code which represented the characteristic functions of various math concepts. It was only because of the deep relationship between LISP and Mathematics that these operations (loop unwinding, recursion elimination, composition, argument elimination, function substitution, etc.) which were basic LISP mutators also turned out to yield a high 'hit rate' of viable, useful new math concepts when applied to previously-known, useful math concepts. But no such deep relationship existed between LISP and Heuristics, and when the basic automatic programming operators were applied to viable, useful heuristics, they almost always produced useless (often worse than useless) new rules. Our work on the nature of heuristics has enabled the construction of a new language in which the statement of heuristics is more natural and compact. Briefly, the vocabulary includes many types of conditions, actions, and descriptive properties that a heuristic might possess; instead of writing a large lump of LISP code to represent the heuristic, one spreads the same information out across dozens of 'slots'. By employing this new language, the old property that AM satisfied fortuitously is once again satisfied: the primitive syntactic operators usually now produce meaningful semantic variants of what they operate on. The ties to the foundations of Heuretics have been engineered into the syntax and vocabulary of the new language, partly by design and partly by evolution, much as John McCarthy engineered ties to the foundations of Mathematics into LISP. The EURISKO program embodies this language, and it is described in this paper, along with its results in eight task domains: design of naval fleets, elementary set theory and number theory, LISP programming, biological evolution, games in general, the design of

Artificial Intelligence **21** (1983) 61–98

three-dimensional VLSI devices, the discovery of heuristics which help the system discover heuristics, and the discovery of appropriate new types of 'slots' in each domain. Along the way, some very powerful new concepts, designs, and heuristics were indeed discovered mechanically. Characteristics that make a domain ripe for AM*-like exploration for new concepts and conjectures are explicated, plus features that make a domain especially suitable for* EURISKO*-level exploration for new heuristics.*

1. Design Decisions in Constructing the EURISKO Program

Our earlier papers in this 'Nature of Heuristics' series [9, 12] have motivated the task of the EURISKO program: learning by discovery, in particular learning new heuristics as well as new domain-specific definitions of concepts. They have given little attention to the architecture of that program, to its results, or to what—in hindsight—they reflect on our earlier experiences with AM [3]. The EURISKO project was first conceived in 1976. During the past six years, there has been an accumulation of 'design ideas' which have been tested. Some of these have been built into the representation language underlying EURISKO (i.e., RLL [8]). Other ideas have found their way into the EURISKO knowledge base itself, as explicitly represented (and malleable) concepts. This section presents these ideas and design decisions, and in Sections 2 and 3 we discuss the performance of the EURISKO program. The design ideas fall naturally into three categories: those dealing with representation, with control, and with the user interface.

1.1. Ideas about representing concepts

(1) *Rules need not distinguish 'slots' from 'functions'.* As in AM, EURISKO's basic representation employs frames (units) with slots. Each slot can be viewed as a unary function which is handed a unit-name and returns a value. E.g., Worth and IsA are slotnames; they are the names of properties a unit might possess. But they can also be considered unary functions: Worth(SetUnion) = 650; IsA(SetUnion) = {SetOp, BinaryOp, Dom = RanOp}. Other unary functions exist, of course, and can be defined in terms of these more primitive slots. For instance, suppose we define AllIsAs as a function which returns the IsA value for a concept, plus all *their* Generalizations, plus all *their* Generalizations, etc. So AllIsAs(SetUnion) first accesses the IsA slot of SetUnion, and finds {SetOp, BinaryOp, Dom = RanOp}. It next looks on the Generalizations slot of those three units, and finds (coincidentally) that they all say {Operation}, so it adds that value to the growing list. Continuing, it finds that the Generalizations slot of Operation contains {Active}, and finally the Generalizations of Active is {Anything}. The final value returned is therefore the set {SetOp, BinaryOp, Dom = RanOp, Operation, Active, Anything}. The point here is that the system's heuristic *rules* can refer to Isa(SetUnion), and they can refer to AllIsAs(SetUnion), and they need never know nor care whether one or both of them are primitive slots, or in fact whether they are both computed via some more complex algorithm. The decision about which functions are implemented

as primitives (slots), and which are computed dynamically from others, is invisible to the rules, and may change from time to time (e.g., after a great amount of experience is accumulated in some domain, it may be apparent that AllIsAs is requested so often that it should be stored primitively). The rules represent guidance knowledge which is, after all, independent of the specific representation being employed; this 'slots = functions' scheme adequately decouples the two. From the point of view of the rules, all a 'concept' is is a legal argument for a list of functions (mostly unary ones).

(2) *'GET' knows why it's being called, 'PUT' knows how the value is justified.* As in most frame-based systems, the most fundamental access functions are GET and PUT, rather than, e.g., ASSERT and MATCH. The above paragraph shows that instead of writing (GET *C f*), which would mean "get the value stored in slot *f* of concept *C*"), we shall write simply *f*(*C*). It became painfully obvious during the building of AM that GET was being called for several different reasons in different places. Sometimes, all that was wanted was to know if *any* values at all were known yet for *f*(*C*); sometimes an AM rule wanted to know the *length* of the set of values; sometimes it wanted to know *some* values, but it didn't matter how up-to-date the answer was; often the major constraint was a limitation on the amount of *resources* to expend (time or space or number of queries to the human user); and occasionally 'the complete answer' was required, regardless of how difficult it was to obtain. In EURISKO we have begun to accomodate these different reasons and constraints on each call on GET, by providing extra arguments which specify which reason is behind this call (Existence, Length, Some, Up-to-date) and how much resources can be spent (Time, Cells, Queries). Calls on PUT are more standard; they may trigger some flurry of re-writing, but the only extra argument one wishes to supply is an indication of the *justification* of the value being changed. For example, was this value computed by using values obtained by GET? The answer is almost always affirmative, so one then asks just how precise *those* values were; e.g., if they were all obtained under severe time limits, the value we're about to PUT will be of dubious accuracy.

(3) *'The size of ships' can mean different things, and there should be a place for each.* Consider what it means to say that the Size(Ships) = Large. We can find many separate interpretations; here are half a dozen:

(i) Each ship is large (this is guaranteed).

(ii) The default answer, when asked how big a ship is, is 'Large' (but no guarantee).

(iii) The EURISKO units representing ships take up a lot of memory.

(iv) There are many elements in the set of all ships.

(v) Looking over the unit representing the set of all ships, we see it is very big.

(vi) Viewing Ships as the name of a kind of slot (e.g., a unit representing a fleet might have a slot called Ships, which was filled with a list of ship-names), we note a very large number of entries on such slots (for those units which can have a Ships slot).

Our aim here is not to argue for a preferred meaning, but rather to point out that each meaning should be unambiguously representable. Having ambiguity in English adds flair to the language, and makes conversation exciting; but it makes computation much more costly. We opt to eliminate ambiguity at the time knowledge is entered into the system, so that, e.g., the way that Size(Ships) = Large gets entered is guaranteed to disambiguate among the six meanings listed above. In brief, we (a) replace a unit for X with separate units for AllXs, TypicalX, and XquaSlot, and (b) have meta-knowledge recorded in slots that are prefaced 'My'. Eliminating redundancy internally reduces processing time required, at a small increase in space required. Preserving meta-knowledge uses quite a bit of extra space, but enables heuristics to later perform inductions that would otherwise be impossible (e.g., noticing regularities in all units entered by a certain person, or all attempts to synthesize examples by a certain new heuristic).

(4) *Each kind of slot has a unit describing it.* In building a knowledge base, the need arises to be able to say things *about* each kind of slot. We give three examples.

(a) What does it mean when a value is stored in slot *s* of unit *U*? Is that value *guaranteed* to be a legal entry, or is it just *probable* that it belongs there? Is it going to *always* be valid, or is it merely *currently* a valid entry? These and other questions about the epistemological status of entries on a slot will change from one task domain to another, from one program to another, from one slot to another. Each kind of slot should 'know' what it means to have a value stored on itself.

(b) Another question whose answer will vary is: Should we redundantly cache (store) this value, or just assume we'll recompute it whenever we need it? Some languages, such as the MOLGEN units package, force the answer to always be 'redundantly store'; most languages force the answer to be 'redundantly compute'. But the optimal answer will depend on how the knowledge base grows, changes, and is used (e.g., how often are the values accessed, compared to how often they're changed? How much space do they take up?)

(c) When an entry is added or removed from an IsA slot, we expect the 'inverse link' to be likewise added or removed. This could be built in for each type of slot, but that makes defining new slots hard for the user.

These examples illustrate the utility of representing each kind of slot as a unit. Thus there are units called CompiledCode, Generalizations, IsA, etc. For instance:

NAME: IsA, Isa, Is-a, ISA, IS-A
  Informally: is, element-of, is-in
DOMAIN/RANGE: ⟨Units → SetOfUnits⟩
IS-A: Sot
FilledWithA: Set
EachEntryMustBeA: Unit representing a set
Inverse: Examples
UsedByInheritanceModes: InheritAlongIsAs
MakesSenseFor: Anything
MyIsA: Eurisko unit
MySize: 500 words
MyCreator: D. Lenat
MyTimeOfCreation: 4/4/79 12:01
Generalizations: AKindOf
Specializations: MemberOf, ExtremumOf
Worth: 600
Cache: Always
English: The slot which tells which classes a unit belongs to.
ALGORITHMS:
  Nonrecursive Slow PossiblyLooping: λ (u) $\{c \in \text{Concepts} \mid c.\text{Defn}(u)\}$
DEFINITIONS:
  Nonrecursive Fast PossiblyLooping: λ (u,c) c.Defn(u)

Most of the slots present for IsA were also meaningful for, e.g., SetUnion [3], but a few are new and worth commenting upon. The Inverse slot is filled with Examples; whenever x is added to (removed from) the IsA slot of y, y will be added to (removed from) the Examples slot of x. The MakesSenseFor slot is filled with Anything. This slot describes the class of concepts that can legally have an IsA slot—in this case any concept at all. MakesSenseFor(Domain/Range) = Active, since only active concepts (those with algorithms) can have domains and ranges. Naturally MakesSenseFor(MakesSenseFor) = Slot, since no other type of unit can legally have a MakesSenseFor slot. The Cache slot of IsA says Always; it might have said Never (which would save some space and squander much time) or some more dynamic predicate instead.

1.2. Ideas about control (agendae, reasons, and heuristic rules)

(1) *The control structure of the system is represented as part of the knowledge base.* While an AM-like agenda mechanism has been retained, the precise control algorithm is represented within EURISKO as a set of concepts, so the system can modify it itself. Basically, there is Select-Execute-PostMortem loop represented as a unit. Specializations of this unit form the three nested loops that characterize the EURISKO program: select and work on a topic; given a topic, select and work on a promising task; given a task, select and obey a relevant individual heuristic rule. Each topic is a major category of investigation for the program

(e.g., Number Theory, Device Physics, Games, Evolution, Oil Spills); each task is an order of magnitude more specific and minuscule (e.g., "Find some examples of prime palindromes", "identify the functionality of a newly-designed VLSI device"), and the execution of individual heuristics are yet another order of magnitude smaller ("if trying to find extreme examples of *C*, then extract the base step from a recursive definition of *C* as one such example"); and see the dozens of heuristics R1–R26 discussed in [12]).

Representing the control structure explicitly has had three benefits so far. First, it facilitates explanation; EURISKO can more coherently explain what it is doing at any given moment, when asked by the user. Given that it's running in a particular function, the user can ask what the purpose of the function is, how long it usually takes to run, why it was called, etc. Second, it allows *enforced semantics* [8]. Given that individual rules are supposed to take about a minute to run, that an IfPotentiallyRelevant test is supposed to be much faster than an IfTrulyRelevant test for each rule, and other assumptions upon which the system has been built and optimized, it now has a way to enforce those constraints. If, for instance, a rule is created whose IfPotentiallyRelevant is taking longer than its IfTrulyRelevant, explicit representation and record-keeping of the control structure will let this be noticed and corrected. This situation has happened many times, for rules synthesized by other heuristics; it might well happen in the future when new human users begin adding rules to the system with only a partial view of what the various slot names are supposed to mean. Third, and finally, since the average time and space for each function (and the variances of time and space) are built up over a reliable sample of cases, it is possible for EURISKO to notice when it's in danger of being in an infinite loop, even a subtle one in which there is no obvious infinite recursion or circular list structure involved. The original motivation for explicitly representing control was to enable the program to meaningfully modify its own control code, but this has always resulted in bugs (due to an inadequate mastery of programming, of models of learning, and so on).

(2) *Multiple agendae.* The human researcher sticks with a topic for an extended period of time. Partly this is due to the difficulty of 'swapping in' a whole new set of concepts, heuristics, etc. Yet part of the reason for this behavior is more rational, and worth duplicating in our mechanical researchers: a developing field will often bog down and appear to stagnate, and this gradual winding down of interestingness is punctuated by occasional bursts of (often serendipitous) discovery, which lead to many promising things to do, which gradually wind down, etc. If the human—or the machine—abandons a topic as soon as it begins to level out, he/she/it will forever be limited to making very superficial discoveries in many fields. How, then, does EURISKO stay focused on a single topic for a nontrivial period of time?

Some (initially eight) of EURISKO's concepts (e.g., Games, DevicePhysics,

NumberTheory) represent 'topics'. Each topic has a slot called Agenda, which contains its own agenda of tasks dealing with that topic (concept) and/or with one or more of its specializations. There no longer is one central agenda; rather, there is a 'current topic', and *its* agenda is the one being used for a while.

When a task is proposed which deals with a concept *C*, EURISKO ripples up from *C* along the Generalizations links looking for topics, halting as soon as it finds one. Since a concept may have several immediate Generalizations, there may be several upward ripplings going on at once; each one terminates as soon as it finds a topic. For instance, suppose some rule proposes a new task involving Palindromes. Their immediate generalizations are Numbers and SymmetricConstructs. These eventually lead to the topics of NumberTheory and Aesthetics. A pointer to the task is put on the agenda of each topic encountered. There can be several pointers to the same task simultaneously existing on different agendae.

Below we examine the mid-level loop (choosing tasks and working on them) and low-level loop (choosing heuristics and obeying them). Here we are considering the *top-level* loop, which involves choosing a topic, working on it for a while (minutes to—rarely—days of CPU time), and then performing a post-mortem (after which the loop repeats). Once a topic is chosen, the next lower level of loop is entered: choose a task, work on it, and analyze what happened. Note that a user's interests (as defined by the concepts that model individuals and groups of individuals) may affect which topics EURISKO expects the user to be interested in, which tasks he would most like to see worked on, etc.

(3) *Dynamic creation and elimination of agendae* (*topics*). On rare occasions, a heuristic rule will advise that an agenda be split into pieces. E.g., here are two rules which make such recommendations:

> **If** agenda *A* contains more than four times as many tasks as the average agenda,
> **then** (try to) split *A* into about three pieces.

> **If** the number of units called on per task, when working on tasks of agenda *A*, is more than ten times the rate at which other agendae inspect units,
> **then** (try to) split *A* into two pieces.

Once the recommendation is made, other rules have some ability to meaningfully effect such schisms. One easy way to do this is by creating a new agenda for each specialization of the concept (=topic) of the original big agenda *A*. If there are no known specializations of that concept, other rules may still apply. One rule looks for groups of concepts mentioned in some fraction (ideally $\frac{1}{3}$) of the tasks on *A*, but on very few (ideally less than $\frac{1}{10}$) of tasks on other agendae, and then uses these groupings to delineate the few new topics. Each new topic is explicitly defined and marked as being a new specialization of *A*, and the tasks from *A* are parcelled out onto the new, specialized, smaller agendae.

When an agenda shrinks too small, rules cause it to be merged into all appropriate immediate generalizations' agendae. In such cases, the general agendae should adopt (a little of) the small agenda's aesthetics, values, heuristics, reasons, goals, open problems, points of view... In practice, so far, the only things inherited are the tasks themselves. This raises a possible research question, but is not currently an area we are investigating.

In one run, EURISKO split the Games agenda into two pieces: one dealing with the Traveller fleet design game, and one dealing with all other games. As it ran out of things to try in the Traveller domain, that agenda grew shorter and finally (days later) was automatically reincorporated into the Games agenda.

(4) *Selecting a task: the half-frame problem.* Let's look in more detail at the three phases for the middle level: selecting a task, finding rules which help satisfy it, and doing a post-mortem on the aftermath on the task's execution. Selecting a task is done as follows. The top task's reasons are evaluated carefully, and its rating is updated. Reasons often become stale, but rarely (during this phase) does a new reason suddenly spring to mind; therefore, a task's rating will almost never increase, but may decline quite a bit by the time we get around to it. We term this the 'half-frame problem' because it reminds us of McCarthy and Hayes' [13] frame problem, but in a world where changes go only in one direction. That constraint lets us efficiently 'solve' the problem: If, after reevaluation, the top task's rating falls below that of task number 2, we merge it back into the agenda, and repeat this step. Finally, some task will stay at the top of the agenda (or we'll be down to re-re-evaluating some task which was higher initially—hence we know it won't be lowered any further). One way or another, then, this phase terminates by selecting a task from the agenda. Not all the lower-rated tasks have been reevaluated at this time, but that doesn't matter because reevaluating them would only have lowered their ratings anyway, so they *almost certainly* aren't the top task to work on now.

(5) *Executing a task: dynamically assembling a rule interpreter.* The second phase then begins. The first activity is to locate a set of potentially relevant heuristic rules, rules whose execution may (help to) satisfy the chosen task. Space and time bounds are computed (and may be updated as the rules fire). Executing a rule is not so straightforward as it was in AM or most other rule-based systems. There are several ways that the pieces of the relevant rules can be run as executable code—i.e., several possible rule interpreters. Various parameters of the current situation determine which rule interpreter is used. Here are nine examples.

> If resources are quite limited, the conditions If-Enough-Time, If-Enough-Space, If-Enough-UserAccess, etc. will be checked quite early on.*

If the current task is vitally important, those slots may never even be considered.

If very few rules are potentially relevant, then there's little need to spend time ordering them.

If very many rules are potentially relevant, it may be better to evaluate some If slot of *all* of them, and then place them in some order for further, detailed consideration of relevance.

In some particularly tricky situations, the rule interpreter must know that it is only a provisional choice, and that it must look for features of the environment (as it runs) that cause it to suspend, and initiate a quest for a better interpreter.

A more common, and less drastic, situation occurs when a rule interpreter knows it must occasionally check for some new rules which might have become relevant since the start of the rule-executions.

All other consideration being equal, prefer a specific rule to any of its generalizations, prefer a rule with shorter running time, with (average) fewer number of user interactions initiated, with higher Worth, with more ancient TimeOfCreation, etc.

If the user is impatient (according to the user model, which, e.g., might have noticed a flurry of ↑T's being typed), then execute the ThenPrint actions of the relevant rules before actually working on the other Then slots of any of them.

If the user likes conjectures, then execute all the ThenConjecture slots first.

These and other judgmental rules guide EURISKO in choosing and changing—or on some occasions synthesizing a new—rule interpreter for the current task. Once assembled, it is handed control and it runs the potentially relevant rules. This is itself usually a select-execute-analyze loop, which proceeds until the resource bounds have been exceeded, or the rules have all quit.

(6) *Post-mortem of a task*: *non-blind* '*suspend and resume*'. After the second phase ends, a careful analysis is performed upon that activity. What happened? How many rules succeeded? How long did the task take? How much space? Is the user (as represented by units modelling him and the groups he belongs to) still interested in this topic, or is it time to (possibly) switch to a new one? The task is re-examined in light of its reasons: is it now worth putting back on the agenda? With what reasons?

If a task failed, it will usually be placed back on the agenda along with some new tasks which (if they succeed) might enable this one to run successfully. This task's failure serves as one reason for those new tasks, and when they succeed *their* post-mortem should boost the priority of this task. In trivial cases (where no heuristics know why the task failed, what could have helped it), the task is simply put back on the agenda, and this mechanism resembles the familiar *blind* suspension and resumption of processes. What Eurisko's control structure allows here is a sort of *best-first* knowledge-guided generalization of that mechanism.

(7) *Each heuristic rule is itself a concept; we do not distinguish metarules from rules.* Each heuristic rule examines some concepts, modifies them, creates similarly typed ones, etc. It just may so happen that some of those concepts examined and synthesized will be heuristic rules (will themselves be capable of operating upon other concepts). For that matter, they might even be concepts that are heuristics which work on heuristics; this was illustrated toward the end of Section 4.12 of [12]. There is no need to distinguish metarules from rules, as we can now simply apply a body of heuristics to itself as well as to concepts from some technical task domain. After all, categories should be drawn when and only when the distinctions lead to some advantage, some new ability or clarity or power.

Unfortunately for my philosophy, EURISKO recently chose to define and separate out the set of rules that can operate sometimes on other rules—i.e., the metarules. It did this mainly for aesthetic reasons (co-identification), and decided to keep the distinction around because it noticed a powerful regularity involving metarules: running one on rules usually takes much longer than running it on domain-level concepts. In hindsight it's clear that testing a rule will take an order of magnitude more effort than testing an object-level construct, because testing a rule might require synthesizing and testing a dozen domain concepts along the way. Nevertheless, each metarule can and does still run at both levels.

As part of what we get from representing each rule as a full-fledged unit, the rules are automatically now organized into an enormous generalization/specialization hierarchy. The so-called Weak Methods (generate and test, hill climbing, etc.) lie at the top (most general), and there are many hundreds of entries near the bottom (specific judgmental rules which mentioned particular terms like 'n-doped', 'nuclear dampers', and 'perfect numbers'). But what of the structure in between? In particular, what are the next hundred or so nodes below the five weak methods? What is the average depth of the tree, the average branching factor, and so on? One aim of this research is to get a better grasp of what this 'space of heuristics' looks like, what its structure is. Our results to date were recently presented in [9]. Even though the weak methods are encoded as rules in EURISKO, and can be run, few of them have ever

succeeded in producing a useful result, and as a result their Worths are fairly low. It is usually more efficient to devise for the program a specific heuristic for some new situation, rather than spending the extra time following a very general heuristic.

Once a task is chosen, say working on concept *C*, a rule interpreter is chosen or synthesized. This is run on the set of potentially relevant rules, namely the rules pointed to by *C* or by one of its generalizations. The organization of rules into a tree enables this set to be small (on the order of the log of the total number of rules in the system). The interpreter will evaluate If parts of rules and execute Then parts in some fashion (perhaps dealing with rules one at a time, perhaps running all their If's and then picking a rule at a time to carry out its Then's, perhaps running all the Then-Conjecture slots of all truly relevant rules immediately, etc.) The post-mortem of an individual rule is necessarily simple: bookkeeping information about time and space used, new units created, etc. are recorded.

1.3. Ideas about communication

(1) *As* EURISKO *matures, it interacts less as a pupil, more as a co-researcher.* As with all expert systems, much system-user interaction has been required initially, at system-startup time. These dialogues have been primarily tutorial, as we put in one concept after another by hand. Later interactions were less frequent, less tutorial in character, more frequently involving outside experts watching and interacting with the program as a performer. Besides models of users and user-groups, EURISKO should have models of dialogue-modes (tutoring the system, solving problem, being taught by the system, etc.) We did, and still do, believe this to be important, but little work has been done on it as yet.

(2) EURISKO *must quickly notice when new concepts are related to existing ones.* EURISKO generates new concepts frequently. One result we've noted is the high frequency with which these 'new' concepts are in fact equivalent to an already-existing one. So EURISKO should have a fast way of checking each new concept, to see if it's genuinely new or not. We call this 'the recognition problem'. It arises both when the user defines some new concept, and when EURISKO itself does.

EURISKO currently employs the following strategy to deal with this problem. Each unit knows which slots are criterial, i.e., define it. Each such criterial slot *s* knows the way in which it makes sense to do matching. The existing concepts caught by this simple mechanism can then be examined in detail at leisure. For instance, a concept may have Defn as a criterial slot, and the Defn may be a conjunction of tests. Consulting AND, the matcher finds that it is supposed to be insensitive to the order of the conjuncts, and that it should recur on *their* structure to determine if they match. Another concept has a criterial slot which is Alg (a procedure for computing some function). The

Alg slot is filled with a LISP PROGN, and the matcher consults PROGN and finds that EVAL-ing the program on test arguments to see if gives the same answers is one way of testing a match. This does not provide a theoretical solution to the general problem of finding potentially-related concepts, but it is working satisfactorily, empirically.

(3) EURISKO *always has the initiative*; *the user can request but never demand.* When the user types in some message indicating that he wishes to define or modify a concept, that request is placed as a very (but not infinitely-) high priority task on the agenda. Note that EURISKO does not relinquish control at any time; it keeps the initiative. When the user types in 'interrupts', EURISKO does minimize the amount of time until his/her interrupts are handled, but that is only due to courtesy (as defined by rules), not built in to the system in any way.

When the user selects a topic, that topic is given much greater weight than any others; yet there may well still be some task on some agenda which has so high a rating that its done anyway. The model of the user (based on him/her as an individual and also based on groups the user belongs to) determines how to treat his/her requests and interrupts. Some categories (such as AI researchers) enjoy seeing a program retaining full control; for other groups (such as mathematicians), EURISKO knows it must (and does) simulate being a quite subservient program.

(4) *Modelling the user enables the creation of a good first impression.* Creating a good 'first impression' is important. Psychologically, it will overshadow the user's attitude toward using the system for a long time to come. Pragmatically, EURISKO is dependent upon outside experts for testing, use, and knowledge base building; it is important to keep them interested in interacting with the program. Telling a user something he/she already knows about, or omitting an explanation in an area he/she is unfamiliar with, are equally serious turn-offs.

EURISKO solves this problem by building up and using models of its users. When a new user logs in, the program attempts to quickly guess as much as possible his profession, his interests, his notations. Many of these features are co-occuring (e.g., if he writes 'j' to indicate the square-root of -1, then he's probably an engineer, and he'd feel at ease being shown huge equations and formulae). Thus, when a few things are observed, EURISKO can tentatively assign (as defaults, as it were) all the other known co-occuring 'symptoms'. This kind of expectation-filtering inference forms the common source of power for many current AI methodologies (frames, scripts, beings, stereotypes, units, schemata).

To support expectation-filtering by user models, a massive data base must exist, dealing with people in general, broken into groups, and even some data about specific known individual users. As one might expect from the EURISKO philosophy, each person, each group, and the set AllPeople, get their own

separate full-fledged concept frames. A genl/spec hierarchy exists, with the most common knowledge at the top node ('AllPeople' in general), and the most specific knowledge at the bottom nodes ('Polya', 'Feigenbaum'). At present there is not much in this knowledge base, but the results it affects are often the most noticeable ones by outsiders. For instance, when a new user types ↑T as EURISKO is starting up, EURISKO concludes that he/she is familiar with other computer systems, is impatient, is probably scientifically-oriented, etc. EURISKO does learn simple models of each new user, but there are at present very few psychological and societal heuristics for building up (and testing!) such models. Based on our model of theory formation (presented in [12]), we are not surprised that only minimal sorts of learning were achieved without a deep model of the domain (in this case, the domain of 'the psychology of using computer programs').

2. Results of EURISKO Applied to Naval Fleet Design

New concepts lie 'near the surface' of all fields, though of course some fields have been picked cleaner than others (e.g., contrast number theory to AI). Mathematics was a poor choice of domain for AM from this point of view, since it has been so well explored throughout the millenia. It is rare that interesting new results arise near the surface of old disciplines; one exception is Conway's numbers [7]. In fact, in our first graph theory protocol, such a concept was discovered (the category of graphs now known as uniquely geodesic: if a path exists between two vertices, then a unique shortest path exists). In AM, there was always the possibility that while each heuristic seemed intuitively obvious and general, its true nature was merely an encoding of some of known mathematics, and that that was in fact why it appealed to our intuition (that our intuition has been shaped to reflect a rough image of mathematics that exists already). We strongly believed this *not* to be the case, however. EURISKO has been a good test of the hypothesis that a large but general set of judgmental rules for manipulating concepts (and for discovering new rules) can be found and operationalized.

To demonstrate the efficicacy of its methods to practitioners of the fields it works in (e.g., mathematicians) and to practitioners of AI, any program claiming to be a 'discovery program' should aspire to two goals: (i) use the same methods to discover concepts, conjectures, and heuristics in several domains, and (ii) make at least a few genuinely new (to mankind) useful discoveries. AM did not meet these criteria well, but EURISKO does. Sections 2 and 3 of this paper discuss the various tasks EURISKO has worked on, and the results it has achieved. Other articles focus on applications to VLSI design [11], elementary mathematics [3], and biological evolution [10]; therefore we shall concentrate upon a different task, that of designing a futuristic fleet to compete in the Traveller Trillion Credit Squadron (TCS) wargame [14].

EURISKO designed a fleet of ships suitable for entry in the 1981 and 1982

national Origins tournaments for TCS. These tournaments are held on July 4 weekends, and run by Game Designers Workshop, which is based in Normal, Illinois. The 1981 tournament was held in San Mateo, CA, and the 1982 tournament was held on the campus of the University of Maryland, in Baltimore. Each tournament was single elimination, six rounds. EURISKO's fleet won the first tournament, thereby becoming the ranking player in the United States (and also, an honorary Admiral in the Traveller navy). This win is made more significant by the fact that no one connected with the program had ever played this game before the tournament, nor seen it played, and there were no practice rounds. Subsequent to that event, changes were made in the tournament rules, changes which nullified most of the unusual features of the submitted fleet. A different collection of rule synergies were opened up by the new rules, however, and EURISKO's new fleet won the 1982 tournament as well.

Each participant has a budget of a trillion 'credits' (roughly equal to dollars) to spend in designing and building a fleet of futuristic ships. There are over one hundred pages of rules which detail various costs, constraints, and tradeoffs, but basically there are two levels of variability in the design process:

(1) Design an individual ship: worry about tradeoffs between types of weapons carried, amount of armor on the hull, agility of the vessel, groupings of weapons into batteries, amount of fuel carried, which systems will have backups, etc.

(2) After designing many distinct kinds of individual ships, group them together into a fleet. The fleet must meet several design constraints (e.g., *some* ships in the fleet, having a total fuel tonnage of at least 10% of the total fleet fuel tonnage, must be capable of refueling and processing fuel), and in addition must function as a coherent unit.

To handle this task, 146 units were added, by hand, to EURISKO. We list their names below, illustrate a couple in detail, and then discuss what EURISKO did.

Accel AccelAttackInfo AccelUSP Agility Alg Armor AttackInfo AttackInfoOf BeamDefense BeamLaser BeamLaserAttackInfo BigAccel BigAccelDamage BigAccelUSP Bridge DestroyedDamage Computer ComputerDestroyedDamage ComputerFib ComputerInternalDamage ComputerRadiationDamage Config Configuration ConfigurationDefense CrewDamage CriticalHitDamage CriticalTypeOfDamage Damage DamageInfo DamageInfoOf DamageTableOf DefendsAs DefendsUsing DefensiveWeaponType EnergyGun EnergyGunAttackInfo EnergyGunUSP Fleet FleetBattle FrozenWatch FrozenWatch/TroopsDeadDamage FuelDamage Game GameConcept GameObj GamePlaying Games H61 H62 H63 HEMissile HandleComputerInternalDamage HandleComputerRadiationDamage HandleCrewDamage HandleCriticalHitDamage HandleDamage HandleFuelDamage HandleJumpDamage HandleManeuverDamage HandlePowerDamage HandleScreenDamage HandleWeaponDamage Hangar/BoatDestroyedDamage IfDecidingTermination IfSimulating Juggernaut JumpDamage JumpDestroyedDamage LaserUSP ManeuverDamage ManeuverDestroyedDamage Manu MesonGun MesonGunAttackInfo

MesonGunDamage MesonGunUSP MesonScreen MesonScreenDefense
MesonScreenUSP Missile MissileAttackInfo MissileUSP Missler NAccels
NBigAccels NEnergyGuns NLasers NMesonGuns NMesonScreens NMissiles
NNucMissiles NNuclearDampers NNumber NRepulsors NSandCasters
NSmallAccels NucMissile NucMissileDamage NuclearDamper
NuclearDamperDefense NuclearDamperUSP OffensiveWeaponType
PhysGameObj PlayTravellerFleetBattle RangeDesired Repulsor RepulsorDefense
RepulsorUSP SandCaster SandCasterUSP SandDefense Scooter Screen
ScreenDamage ScreenDestroyedDamage Ship ShipVaporizedDamage Ships
SimulationHeurs SizeMod SizeUSP SmallAccel SmallAccelDamage
SmallAccelUSP SmallWeaponDamage Spine/FireControlDestroyedDamage
TerminationHeurs ToHit ToHitLongRange ToHitShortRange ToPlay ToPlayOf
Tonnage TravellerFleet TravellerFleetBattle TwoPersonGame TypeOfDamage
USPNucMissile UnFairGame UsedInSimulating UsedInTerminating UspPresent
WarGame Weapon WeaponAttackInfo WeaponDamage WeaponType

Each concept is represented by a unit that takes about half a page of text to describe. Here are two of these units, shown the way they exist in EURISKO. The first describes a type of weapon available to ships; its offensive power is described in more detail by the second unit.

Name. EnergyGun	**see Note**
Generalizations: (Anything Weapon)	
AllIsA: (GameConcept GameObj Anything Category WeaponType DefensiveWeaponType OffensiveWeaponTypo Obj AbstractObj PhysGameObj PhysObj)	1,3
IsA: (DefensiveWeaponType OffensiveWeaponType PhysGameObj)	1
MyWorth: 400	2
MyInitialWorth: 500	2
Worth: 100	2
InitialWorth: 500	2
DamageInfo: (SmallWeaponDamage)	
AttackInfo: (EnergyGunAttackInfo)	8
NumPresent: NEnergyGuns	3
UspPresent: EnergyGunUSP	4
DefendsAs: (BeamDefense)	4
Rarity: (0.11 1 9)	5
FocusTask: (FocusOnEnergyGun)	6
MyIsA: (EuriskoUnit)	
MyCreator: DLenat	7
MyTimeOfCreation: "4-JUN-81 16:19:46"	7
MyModeOfCreation: (EDIT NucMissile)	7

Name: EnergyGunAttackInfo	
MyWorth: 400	
Worth: 500	
AllIsA: (Anything GameConcept)	
IsA: (GameConcept)	
Generalizations: (Anything WeaponAttackInfo)	
AttackInfoOf: (EnergyGun)	8
ToHitShortRange: (8 7 7 6 6 5 5 4 4)	

```
ToHitLongRange: Impossible
SandDefense: ((4 3 2 1 0 0 0 0 0)
              (5 4 3 2 1 0 0 0 0)
              (6 5 4 3 2 1 0 0 0)
              (7 6 5 4 3 2 1 0 0)
              (8 7 6 5 4 3 2 1 0)
              (9 8 7 6 5 4 3 2 1)
              (10 9 8 7 6 5 4 3 2)
              (11 10 9 8 7 6 5 4 3)
              (12 11 10 9 8 7 6 5 4))
FocusTask: (FocusOnEnergyGunAttackInfo)
MyIsA: (EuriskoUnit ValueOfASlot)
MyCreator: DLenat
MyTimeOfCreation: "4-JUN-81 16:33:18"
MyModeOfCreation: (EDIT MissileAttackInfo)
```

Note 1. The 'IsA' slot holds the immediate (most specific) sets to which EnergyGun belongs; the slot 'AllIsA' holds those same entries, plus all their Generalizations, plus all *their* Generalizations, etc. The 'MyIsA' slot indicates what this data structure is, namely a unit in an AI program.

Note 2. The slot 'MyInitialWorth' records the value of the MyWorth slot of the unit at the time it was created. If the value of MyWorth has never changed, then there is no MyInitialWorth slot needed—such units only have a MyWorth Slot; EnergyGunAttackInfo is such a unit. These values reflect how useful the unit has been to EURISKO—i.e., how compact it's been, how little CPU time it's wasted, how many interesting analogies were built using it, how many of the structural modifications done to it were fruitful, etc. This is to be contrasted with the Worth slot of, e.g., EnergyGun, which specifies how useful energy guns are to have on ships. All values are in the 0–1000 range. What happened to lower the MyWorth of EnergyGun? At one time, it was selected as a candidate for modification; EURISKO spent some time trying to analogize between it and other types of weapons, and nothing much came out of that. As a result, its MyWorth was dropped from 500 to 400. Why was the Worth of EnergyGun lowered? Through many tens of simulations, it became clear that one could buy enough armor plating to make a ship invulnerable to attacks by these types of weapons, and from then on almost all ships were so armored. Thus, any ships having energy weapons were at a serious disadvantage, and gradually—as they lost—the Worth of EnergyGun declined. Incidentally, this mistakenly led to a correct heuristic: "If a weapon cannot hit at all at one range (e.g., Long Range in this case), then it's probably not worth having too many of them." That was *not* the major problem with energy guns, but the heuristic is a good one anyway.

Note 3. Most of the slots are filled in with names of other units; for example the AllIsA slot of the EnergyGun unit points to the unit called WeaponType. If

new facts should emerge about weapontypes (e.g., a heuristic that the bigger the better, or a new regulation affecting the grouping of weapons into batteries), the unit WeaponType would be edited and modified, but there would be no need to try to figure out all the effects on other units in the system. The changes would be inherited by EnergyGun and only noticed when they were needed, when some use was about to be made of them. This is a major source of power in most frame-based systems. Some of these pointers are actually indirect SeeUnit; references [8]. For instance, the NumPresent slot of EnergyGun should be filled simply with a range limit; instead, it contains the list (*SeeUnit:* NEnergyGuns). There is a unit called NEnergyGuns, with a Value slot that does provide the desired range [anywhere from 0 to Tonnage/100], but NEnergyGuns contains other information about the number of energy guns on a ship, such as power requirements, balancing, variance and mean values for this quantity, extreme values, etc.

Note 4. Some slots have a single entry as their value, and some have a list of entries (even though there may only be one element in that list). The unit representing each kind of slot indicates the kind of structure to use to fill incarnations of that kind of slot—set, bag, list, single item, etc. That unit also specifies the type and range of the entries that are permissible thereon.

Note 5. The slot called 'Rarity' reflects the fact that, during a recent run, EURISKO examined nine objects, known to be weapons, to see if they were energy guns; one of them was. This is the kind of bookkeeping record which heuristic rules might want to access; e.g., rules which say "If *C* is a specialization of *G*, and (empirically) very few *G*'s turn out to be *C*'s, then...".

Note 6. As EURISKO worked in this domain, over a hundred new workhorse concepts were trivially synthesized by EURISKO: concepts of the form FocusOnX, where X is one of the 146 concepts. Each such FocusOnX concept represents a task, and is pointed to by one or more agendae, initially the Games agenda. It records the various times that it was the top task and energy was expended working on it, when that happened, what the results were, how long it took, what reasons recommended, it, etc. The FocusOnEnergyGun task thus serves two purposes:

(i) it is a task on an agenda, and when selected it draws EURISKO's attention to the concept EnergyGun; various generation rules might then cause EURISKO to explore modifications to EnergyGun, analogies to it, patterns in its use, etc.

(ii) it is a record of all the times that that task has ever been worked on, and as such forms data which can be examined by rules, e.g., this one: "If most attempts to do X have been slow but fruitful, then...". That is, it serves as data to induct upon.

Note 7. The preface My means "treated as a EURISKO unit, a data structure in a

computer program". So MyCreator refers to the person who first typed in the unit called EnergyGun, or to the name of the heuristic rule which first synthesized it (in which case MyTimeOfCreation would have to point to a task as well as to a date). If the EnergyGun unit had a slot called just Creator, it would be filled with the name of the person who invented the energy gun. Similarly, if it had a slot called TimeOfCreation, it would be the date on which the energy gun was first discovered. Such information might be useful to the program, incidentally, if there were some pattern in the usage of weapons developed by inventor *X*, or developed during a certain period of time.

Note 8. Many of EURISKO's slots have inverses. Thus AttackInfo and AttackInfoOf are inverse slots. IsA and Examples have the same relationship; thus EnergyGun says it IsA Defensive WeaponType. If we ask for Examples(DefensiveWeaponType) the value is the list (BeamLaser PulseLaser SandCaster EnergyGun Repulsor), which includes EnergyGun. On the unit called IsA, its Inverse slot is filled with the entry Examples. The Inverse slot of the Inverse unit is filled with Inverse—so if slot *s* is the inverse of *r*, then *r* is the inverse of *s*.

Of the 146 added concepts, two represented new types of activities: playing a game, and running a simulation. Later, a couple other games were added to EURISKO (Tic-tac-toe and Go) to ensure that the general games concepts truly were general. Managing a simulation caused us to augment EURISKO with three new heuristics (*H*61, *H*62, *H*63); these (respectively) check for termination, try to project the ultimate outcome of the simulation, and check for infinite loops during simulation. One method they employ, e.g., is to monitor the relative strengths of the two opponents; if a pattern develops in the progression of these (e.g., the ratio is always in favor of side 2 and it is increasing in imbalance), a winner can be projected. If the strengths remain basically unchanged for several iterations, an infinite loop (i.e., a draw) can be projected.

Two new If types of slot were introduced, IfSimulating and IfDecidingTermination; the three new heuristics had values for these new slots as well as other, more conventional If slots. Two new categories of heuristics were defined: TerminationHeurs and SimulationHeurs. Their *qua*Slot forms were useful 'unary functions' which map a game—such as TCS—into a set of heuristics for simulating it, and into a set of heuristics for checking termination of the simulation. The inverse links to these two slots are called UsedInTerminating and UsedInSimulating. They point from a rule to the game(s) or process(es) it simulates (or tries to terminate). For instance, heuristic *H*61 has a slot called UsedInTerminating, whose value is the singleton list (TravellerFleetBattle).

The new units collectively specify the rules of the game and the constraints on the design process. How did we get EURISKO to play the game? The unit Games is a topic, and as such can have an agenda. One of the new units,

PlayTravellerFleetBattle, had no examples, so a general heuristic added a new task to the Games agenda: "Find examples of PlayTravellerFleetBattle". The Games topic was selected as the current topic (by pointing to it with a cursor, though it could have been selected indirectly by supplying a user model that claimed the user is very interested in games). Once Games was the chosen topic, there were only a few tasks with higher priority. The first one EURISKO ran defined the difference between Games and TwoPersonGames, and made a note that sometime EURISKO should look into defining some of those Non-TwoPersonGames.

Finally, EURISKO got around to trying to find examples of PlayTraveller-FleetBattle. Each example involved a call on a fleet-designing phase followed by a battle simulation phase. The new concepts and rules helped carry out these processes. After each simulated battle, EURISKO paused to try to abstract from the results some new design heuristics. The first step was to isolate the differences between the two fleets. Often they would be similar, and the differences would exist mostly at the level of design of particular individual ships. Eurisko then framed many different general rules, any one of which would suffice to prefer the winning design over the losing one. No new techniques are used for this induction process; rather, the apparent power is due to a good choice of representation for the rules, one naturally suited to rules, to design rules, and even more particularly to TCS ship design rules. We shall have much more to say about this is Section 5. A fast test of the candidate rules was made, using any relevant recorded battles from the past. If more than one proposed heuristic remained, new variant fleets were designed and simulated, each one embodying one but violating the other heuristics. In cases of circular victories (*A* beats *B* who beats *C* who beats *A*) all the candidates involved were retained, but with somewhat lowered worth. Such situations were interpreted as analogues of local maxima, and EURISKO would try for a very different fleet design for the next iteration.

So fleets fight (each battle taking between 2 and 30 minutes), and the battle is analyzed to determine which design policies are winning, and—occasionally—what fortuitous circumstances can be *abstracted* into new design heuristics. An example of the former (gradual parameter adjustment) was when the Agility of ships gradually decreased, in favor of heavier and heavier Armor plating of the hulls. An example of the latter (fortuitous monsters) was when a purely defensive ship was included in an otherwise-awful fleet, and that fleet *could never be fully defeated* because that defensive ship, being very small, unarmored, and super agile, could not be hit by any of the weapons of the larger nearly-victorious fleet.

EURISKO has by now spent 1300 CPU hours on a personal LISP machine, the Xerox 1100, managing this heuristically-guided *evolution* process. The author culled through the runs of EURISKO every 12 hours or so of machine time (i.e., each morning, after letting it run all night on one or more 1100's), weeding out heuristics he deemed invalid or undesirable, rewarding those he understood

and liked, etc. Thus the final crediting of the win should be about 60/40% Lenat/EURISKO, though the significant point here is that neither party could have won alone. The program came up with all the innovative designs and design rules (i.e., the loopholes in the TCS formulation), and recognized the significance of most of these. It was a human observor, however, (the author) who appreciated the rest, and who occasionally noticed errors or flaws in the synthesized design rules which would have wasted inordinate amounts of time before being corrected by EURISKO.

Most of the battles are tactically trivial, the contest being decided by the designs of the two fleets; that—and the 100-page thickness of the rule-books—were the reason this appeared to be a valid domain for EURISKO. It is also important—for EURISKO to have a good chance of finding new results—that the size of the search space (legal fleet designs) be immense: with 50 parameters per ship, about 10 values for each parameter (sometimes fewer, often an infinite number), and up to 100 distinct ships to design and include in each fleet, any systematic or monte carlo analysis of the problem is unlikely to succeed. In fact, the designers had done a detailed linear programming model of the game, and their computer runs convinced them that a fleet of about 20 behemoths was the optimal design. This was close to the starting fleet design the author supplied to EURISKO, and it was also close to the designs that most of the tournament entrants came up with.

EURISKO was originally supplied with what appeared to be a good fleet design (twenty large ships, each fairly fast and moderately armor-plated, each with some small weapons and one huge spinal weapon). EURISKO also had many 'mutation' operators, such as changing the number of ships, their size, their weaponry, etc. The many constraints—the TCS rules and formulae—were used to constrain the generation of mutant fleets, and to prune away illegal ones before simulating them. At first, mutations were random. Soon, patterns were perceived: more ships were better; more armor was better; smaller ships were better; etc. Gradually, as each fleet beat the previous one (and a few random ancestors), its "lessons" were abstracted into new, very specific heuristics. These rules are specific not only to ship design, but to the particular set of TCS rules in effect during 1981. The design rules were then used to further constrain the mutation process.

One very general result that EURISKO abstracted from this evolutionary design process was a 'nearly extreme' heuristic.

> In almost all Traveller TCS fleet design situations,
> the right decision is go for a nearly—but not quite—extreme solution.

Thus, the final ships had Agility 2 (slightly above the absolute minimum), one weapon of each type of small weapons (rather than 0 or many), the fleet had almost as many ships as it could legally have but not quite (96 instead of 100), etc. Big weapons (enormous spinal mounts capable of blasting another ship to

pieces with a single shot) were gradually phased out, in favor of an enormous number of small missile weapons. The fleet had almost all (75) ships of this type though there was one ship which was small and super agile and purely defensive (and literally unhittable by any reasonable enemy ship), and a couple monstrous hulks which had no chance of defense against *normal* ships, but which had weapons just barely accurate enough to hit any enemy ships that were (of course!) small and agile and purely defensive.

Some of the strangest elements of the final fleet were discovered accidentally rather than as the result of a long, continuous evolution process. The usefulness of a tiny defensive ship was apprehended after a 'lifeboat' was the only survivor from one side's fleet, yet round after round it could not be hit at all. That design was immortalized into a design strategy ("Include one such ship in your fleet!"), and a very general rule began looking for ships that *could* destroy it. Finally, one was found; it was quite strange, and would never have been included except to counter the possibility that the enemy might have small defensive ships too. Against any normally-armed ship, it would quickly be destroyed. Basically, this new ship had moderate size, no armor, the largest possible guidance computer, the slowest possible engines for its size and equipment, and one single, enormous accelerator weapon—a weapon usually ignored because its broad beam glances harmlessly off large armor-plated ships, but which is very easy to aim. This combination is ineffective for most combat, but is just enough to fire at the little boats it might be sent against. We were a little disappointed that none of the other entrants had small defensive "stalemate guarantors" of the sort we took.

Almost all the other entrants in the final tournament had fleets that consisted of about 20 ships, each with a huge spinal mount weapon, low armor, fairly high agility, and a large number of secondary energy weapons (laser-type weapons). This contrasted with EURISKO's fleet in almost all ways. Most ships in our fleet *did* sprout one solitary laser among their 50 or so weapon batteries, but not because it was useful in combat—just to absorb damage from enemy fire (thanks to the somewhat unrealistic scheme by which damage is inflicted on ships which have been hit). After an exchange of fire, most of the enemy behemoths did indeed sink one of EURISKO's ships, for a total loss of about 15 ships. In return, EURISKO's 96 ships sank about 5 of the enemy. So just prior to the second exchange of fire, the enemy was down to 15 ships, and EURISKO 81. After a second round of fire, the numbers were 11 and 70. Two more exchanges brought the totals to 1 and 46, and one more round of fire wiped out the enemy. In this scenario—which was the most common one in all EURISKO's battles during the tournament—there is no need at all to bring any of its specialty ships into the front lines at any time.

The tournament was run in such a way that, after one player wins a battle, his fleet is completely reconstituted and repaired to its original state, in preparation for the next rung of the ladder.

pattern became clear. Its second opponent did some calculations and resigned without ever firing a shot. The subsequent oponents resigned during their first or second round of combat with this fleet. EURISKO's few specialty ships remained unused until the final round of the tournament, battling for 1st versus 2nd place. That opponent also had ships with heavy armor, few large weapons, low agility, etc. He was lacking any fast ships or fast-ship-killers, though. The author simply pointed out to him that if EURISKO were losing then (according to the TCS rules) our side need put only our fast ship out the front line, withdraw all the others and repair them, and—once they were finished repairing themselves—effectively start the battle all over again. This could go on ad infinitum, until such time as EURISKO appeared to be winning, and in that case we would let the battle continue to termination. The opponent did a few calculations and surrendered without fighting. Thus, while most of the tournament battles took 2–4 hours, most of those involving EURISKO took only a few minutes.

The tournament directors were chagrined that a bizarre fleet such as this one captured the day, *and* a similar fleet (though not so extreme) took second place. The rules for future years' TCS tournaments were changed to eliminate the design singularities which EURISKO found. For example, repairing of damaged ships was prohibited, so the utility of the unhittable ship became negligible.

Details of EURISKO's victory at the tournament and a complete listing of the design of our winning fleet are given in [14]. Rules for the competition are given in three parts, each of them necessary, each published in a separate softbound book of about 200 total pages: One on small ship design, one on large ship design, and one on fleet design and combat rules. These are available from Game Designers' Workshop, Normal, Illinois, as well as from game and hobby shops nationwide in the U.S.A.

When rules for the 1982 tournament were announced, EURISKO was set to work on finding a new fleet design. Although many of its best designs and design rules were now illegal or useless, most of the *general heuristics* it synthesized about the game were still valid. Using the 'nearly-extreme' heuristic, for instance, it quickly designed a ship with practically no defense, and that ship filled a key role in the final fleet. Coincidentally, just as the *defensive* ship made a difference in the 1981 final round, the *offensive* ships made a difference in the 1982 final round. In each case, their presence caused the opponent to resign without firing a shot. The bulwark of our 1981 fleet was a ship that was slow and heavily armored; the majority of ships in our 1982 fleet were very fast and completely unarmored. Just as most 'experienced' players jeered at the 1981 fleet because it had practically no large weapons, they jeered at the 1982 fleet because it was unarmored and it *still* had no large weapons, even though the rules changes had made them much cheaper.

What EURISKO found were not fundamental rules for fleet and ship design; rather, it uncovered anomalies, fortuitous interactions among rules, unrealistic

loopholes that hadn't been forseen by the designers of the TCS simulation system. There may be little of what EURISKO found that has application to real naval design; most of its findings pertained to the fine structure of the TCS rules, not to the real world. For example, a crew hit reduces the number of crewmen on a ship from n down to the largest power of 10 smaller than n (e.g., from 370 to 100, from 82 to 10); EURISKO therefore designed ships requiring 99 crewmen, and crewed them with 101 people; the first crew hit therefore had no effect on the ship's battleworthiness.

The fact that EURISKO's discoveries were synergistic loopholes rather than genuine naval insights is not in itself bad, as our goal was to win the tournament, not break new ground in real warfare. In fact, the very unreality of the TCS rules—as any 100-page model of the real world is bound to be incomplete and have rough edges—promised to aid us in our task. Here was a search space that had not been explored much by human beings yet; most designers were applying analogues of rules that hold in real life, and that yielded them reasonable designs—fleets of the kind the TCS people anticipated. EURISKO was able to walk around in the space defined by the set of rules, somewhat awkwardly, but (thanks to its absence of common sense knowledge) with few preconceptions about what an optimal design might be. Perhaps we will know that the program has 'arrived' when it first *fails* to win the TCS tournament. This notion of a large, unexplored search space, not necessarily well-matched to our everyday common-sense intuitions, will come up again and again in the following pages. It appears to characterize those domains for which automated discovery (of both concepts and heuristics) is currently most viable.

The rules will indeed change for July, 1983, including the elimination of drop-tanks (fuel tanks that can be jettisoned; this improves the speed of a ship but may strand it after the battle) and other changes that will force a complete redesign. We look forward to the new challenge.

3. Results of EURISKO Applied to Other Tasks

3.1. EURISKO applied to elementary mathematics

The first domain we added concepts about was mathematics, specifically the same starting collection of finite set theory concepts AM began with. Fifty heuristics were added, which subsumed most of AM's old set of 243. This condensation was the result of joint effort on the part of the author, W.W. Bledsoe, and H.A. Simon, begun in 1978 at Carnegie-Mellon University. Not surprisingly, EURISKO then duplicated many of the results of AM: finding elementary set theory theorems, extreme properties of set operations, and defining useful new objects and operations about 50% of the time. The other 50% of its time was spent about half in generating awful concepts (still a bit

better than AM's hit-rate, though), and half in attempting to produce new heuristics and types of slots. Even though these latter activities were quite rare, they were quite timeconsuming when they did occur. The various set-theory examples presented in [12] were drawn from runs of EURISKO.

The knowledge base now contains number theory (divisibility theory) concepts as well. Most number theory concepts were discovered by EURISKO and later hand-smoothed by the author: e.g., a valid but inefficient factoring algorithm was replaced by an efficient one. It is hoped that EURISKO may find some new results on fringes of that area.

About 200 math concepts were present in the system, to work in set theory and number theory. After about 500 hours of running, another thousand concepts had been considered, and 200 of them had proven interesting (empirically, in the program's judgment, and later confirmed by human inspection). Of these new concepts, 11 were valuable, specific new heuristics, and 7 were useful new types of slots. Of the 7 new slot types, four were slots that only heuristics could possess.

It is worth explaining those four new If and Then types of slots which were synthesized: If-Constant, If-Identity, If-Unchanged, and Then-Conjecture. The three If slots were needed because of the high frequency with which new functions turned out to be closely related to (i) a constant function, (ii) the identity function, (iii) the same function they were synthesized from. After synthesizing the new Then slot, EURISKO defined the two new bookkeeping slots listed above: ThenConjectureRecord and ThenConjectureFailedRecord. Each (unit representing a) heuristic rule, call it H, can have either or both of these bookkeeping slots, as well as having a ThenConjecture slot. The bookkeping slots keep track of how often (for this heuristic H) the ThenConjectureSlot has been evaluated, and what fraction of the time it signalled an error (an Abort message) and forced *H*'s execution to terminate.

Several useful heuristics were discovered by EURISKO. Here is one example:

> "If an inverse function is going to be used even once,
> then it's usually worthwhile to search for a fast algorithm
> for computing it."

This was abstracted from a couple experiences where—in number theory—an operation was very easy to compute, but its inverse took a long time. In particular, Times was quick, but Times-Inv (finding all possible factorizations of a number) was lengthy. The amount of time taken up was large even by comparison to the time required to look for algorithms, so EURISKO produced this heuristic. As with most heuristics, EURISKO would have run better if it had had this heuristic from the beginning.

Sadly, no powerful new heuristics, specific to set theory or number theory, were devised. This may reflect the 'well-trodden' character of elementary

mathematics; i.e., so many great minds have wandered in that territory so long that there is little left to find from shallow experiential induction and analogy. The failure also might be due to the rigid, traditional way in which EURISKO's math knowledge was organized and represented.

3.2. EURISKO applied to LISP programming

Two hundred of the most common INTERLISP functions have been represented as units within EURISKO. This may sound impressive, but falls far short of our original goal, which was to have a separate unit representing each one of EURISKO's functions and any LISP functions they call, giving descriptive information about it. This was to hold for (i) LISP primitives, such as EQ and MAPCAR, (ii) hand coded LISP functions used by the EURISKO system, such as MapUnits and FindRandomSubset, and (iii) implicit unary functions—i.e., all the types of slots a unit might possess, such as MyCreator and IfPotentiallyRelevant. We have only begun to scratch the surface on representing LISP primitives; about one half of EURISKO's own functions have units representing them, and all of the slots have such a unit.

These units about LISP, programming, and EURISKO itself enable the EURISKO program to monitor and modify its own behavior, as well as synthesize and modify new LISP functions. EURISKO gathers data about LISP, just as it did about elementary mathematics or naval fleets.

For example, EURISKO was originally given units for EQ and EQUAL, with no explicit connection recorded between them. Eventually, it got around to recording examples (and nonexamples) for each, and conjectured that EQ was a restriction (a more specialized predicate) of EQUAL, which is true. A heuristic suggested disjoining an EQ test onto the front of EQUAL, as this might speed EQUAL up. Surprisingly (to the author, though not to EURISKO), it did! This turned out to be a small bug in INTERLISP-D, which was then immediately fixed. The bug made EQUAL much slower than EQ in the case of identical arguments passed to EQUAL.

Once it had the conjecture about EQ being a special kind of EQUAL, EURISKO was able to look through its code and specialize bits of it by replacing EQUAL by EQ, or to generalize them by substituting in the reverse order. As the author had been somewhat careful in coding the program, it is not surprising that most of these generalizations were useless, and most of the specializations were downright bugs, but occasional improvements in its own code were made by this policy.

A very general heuristic EURISKO possessed said: "If f can often be used in place of g, and f uses less resources, then replace g by f wherever possible" This was specialized by EURISKO into a new LISP programming heuristic which we recognize as a valid one:

"If you can use EQ instead of EQUAL, do it to save time",

Next, EURISKO analyzed the differences between EQ and EQUAL. Specifically, it defined the set of structures which can be EQUAL but not EQ, and then defined the complement of that set. This turned out to be the concept we refer to as LISP atoms. In analogy to humankind, once EURISKO discovered atoms it was able to destroy its environment (by clobbering CDR of atoms), and once that capability existed it was hard to prevent it from happening.

Its two 'discoveries' were EQ (instead of EQUAL) and NCONC (instead of APPEND); both of these were fast but sometimes wrong (or even hazardous to apply). This led EURISKO to define a class of LISP functions that were fast but dangerous, indeed a useful concept for INTERLISP programmers to form.

Later, EURISKO began to generalize some useful LISP predicates, in some cases predicates we had defined using AND as their central connective. One generalization technique was to remove a conjunct or two, and this often led to errors in evaluation. As a result, one additional interesting LISP heuristic was found:

> "Sometimes 'AND' means 'do in sequence', and sometimes it means 'doable simultaneously', and only the latter case is likely to yield good results if you're considering generalizing a piece of code by removing conjuncts."

EURISKO's progress in this domain was entertaining, and a fundamental feature of this domain became clear: large programs are carefully engineered artifacts, complex constructs with thousands of pieces in a kind of unstable equilibrium. Any sort of random perturbation is likely to produce an error rather than a novel mutant. The analogy to biological evolution is strong. The high 'hit rate' AM enjoyed, mutating LISP functions to find new math concepts, was due to the intimate ties between LISP and mathematics. EURISKO had successes in automatic programming only when it modified functions which had been coded as units. Why was this?

In a unit, each chunk of real LISP code—an entry on a slot of the unit—was quite small and fell into the sterotypical category for that type of slot. For instance, consider the slot called ThenDefineNewConcepts. If a unit (representing a LISP function) had some entries on that slot, one knew exactly what their format would be like (a series of calls on unit-defining and initializing routines), what their purpose was (to bring new units into existence), what kinds of things they were likely to be doing (copying from another unit with some modifications), how long these things should take (about 10 seconds per unit defined), etc. This foreknowledge allowed meaningful changes to be made almost all the time, rather than almost never (in the case of modifying a large, opaque lump of LISP code about which nothing is known).

3.3. EURISKO applied to other tasks

There have been six additional domains in which large-scale applications of EURISKO were done. Below we briefly describe five of these tasks and the

outcomes of applying EURISKO. The sixth domain, OilSpills, relies more on RLL than on EURISKO per se, and is covered in detail in [6].

(1) *Evolution.* One task domain was biological evolution: the simulation of organisms competing, followed by the most fit ones reproducing mutated offspring for the next simulated generation. Heuristics for guiding the mutation process (to increase the viability of the offspring) were easily induced. Some of these were as trivial as noting that whenever an improved animal was produced with a change in parameter X, that animal also happened to have a certain change in parameter Y; this got compiled into the heuristic that in the future any mutations of X ought to have a higher chance to modify Y as well. An example of this from the simulation was "decreased ability to defend in combat" and "increased sensitivity to nearness of predators". An example from *homo sapiens* might be "increased skull size" and "increased cephalopelvic diameter", though it appears our DNA lacks this heuristic.

The net effect of having these heuristics for guiding plausible mutations was that, in a single generation, an offspring would emerge with a whole constellation of related mutations that worked together. For example, one had thicker fur, a thicker fat layer, whiter fur, smaller ears, etc. It is not known whether there is any biological validity to this radical hypothesis, but there is no doubt that the simulated evolution progressed almost not at all when mutation was random, and quite rapidly when mutation was under control of a body of heuristic rules. See [10].

(2) *Games.* A second EURISKO task domain we have not discussed in broad terms yet is that of Games. It was mentioned above, in Second 2, that EURISKO was applied to other games besides Traveller TCS. Indeed, several general Games concepts were added to the knowledge base: material, position, tactic, two-person game, fairness, player, opponent, etc. Also, a few heuristics were inserted, as very general strategies: simultaneous action, feint, pin, trap. These were little used in TCS, since the battles themselves were strategically trivial, or in tic-tac-toe, since the entire search space is too small to warrant that level of consideration. In Go, however, these did get used. The level of play was never very high, but the system demonstrated the application of the general games strategies, and found specializations of them to Go. This is significant because they were derived using Chess and Bridge as model games, by a system builder who did not even know the rules for Go.

One area of current research is getting EURISKO to discover interesting new games; that is, make up a set of rules, simulate the game, and evaluate it according to various criteria (surprise ending, size of search spacing, etc.) The game-independent strategies should be specialized into specific, powerful heuristics by EURISKO. Occasionally, several heuristics which were abstracted from experiences in various games should be generalized into a new high-level strategy. The task was suggested by discussions with Herbert Simon in 1977;

the first parts of the programme were done by Ramano Rao, using EURISKO during the summer of 1981.

(3) *VLSI design.* The most recent task EURISKO has been applied to is that of three-dimensional VLSI circuit design, and the related problem of discovering new physical devices for that technology. This work has been quite successful, and is discussed in [11], so we shall limit our remarks to a couple brief paragraphs here. Technical information about building so-called highrise VLSI chips can be found in [5].

The paradigm for EURISKO's exploration is a loop in which it generates a device, finds its I/O behavior, tries to 'parse' this into functionality it already knows about and can use, and then evaluates the results. At first, we had this loop take place at the level of charge carriers moving through semiconducting material, various types of dopants, electric fields being applied to regions of the plane, materials of different types being abutted, etc. Many of the well-known primitive devices were synthesized quickly, such as the MOSFET transistor and the silicon diode. This is because they were short sentences in the language we had defined (a language with verbs like Abut, ApplyEField, and with nouns like nDopedRegion, IntrinsicChannelRegion).

Our expert, Professor James Gibbons of Stanford University's Center for Integrated Systems, quickly decided that we were working at too low a level, and we switched to the level of conduction paths. The philosophy was that if we could produce an interesting design at that level, he could find a way to realize it in hardware. Our first efforts were systematic searches, and this gave us an appreciation for the size of the search space. A very compact three-dimensional design for a flip-flop was also serendipitously synthesized. We soon switched back into the AM and EURISKO paradigm of using heuristics to guide the synthesis of new devices. Almost immediately, symmetry heuristics produced a very powerful yet simple device, one which simultaneously computes NAND and OR, using only two small metal regions, two n-doped regions, two p-doped regions, and one intrinsic channel region. These devices now form the primitive building blocks of our high-rise chip designs. When stacked into arrays, each device uses only *one* region of each type (n, p, metal, channel). We illustrate the device in Fig. 1, but refer to [11] for a detailed explanation of EURISKO's forays into this domain.

Criteria for interestingness of new devices include nonlinearity, state, computing previously-known function using less space, fewer componenets, less power, faster, easier to produce, etc. A device which is superior along any of these dimensions, even if it is slightly inferior along others, might be useful and is worth naming and saving.

Besides many useful devices and circuits, we now have a few useful heuristics for the task of designing three-dimensional VLSI circuits; in every second metal layer, wires should run N–S (and in the other metal layers, E–W); any 3-D folding of a 2-D design should replace (most of) the pairs of gates sharing a

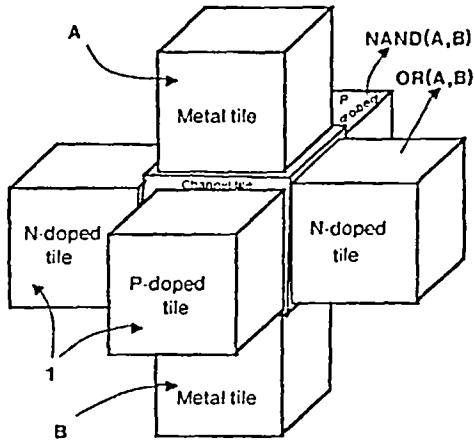

FIG. 1. EURISKO's JMOS cross, the first XMOS device. Side view. When either metal tile is High, a channel connects the two negatively-(n-)doped regions. When either metal tile is Low, conduction occurs between the positively-(p-)doped regions. Note that, if one metal tile is High and one Low, a channel of electrons and a channel of holes both exist and flow (at right angles) inside the central, intrinsic Channel region (nearly obscured in the diagram). The devices tesselate three-space.

common control by single pieces of metal serving simultaneously as gates for regions above and below them; etc.

(4) *Heuretics.* A fourth task domain for EURISKO was that of Heuretics (the study of heuristics) itself. In [9] we discussed the nature of heuristics, why we consider Heuretics to be a scientific field in which one can do experiments and form conjectures, and what we have learned from this work. In operational terms, EURISKO spent time forming and testing heuristics about learning new heuristics. How does it do this? Here is one sequence of behaviors EURISKO carried out:

A heuristic *H*12 had been used many times, and—since heuristics are just special kinds of operators—another heuristic fired, one which said "If an operator has been used many times successfully, it's worth trying to generalize it". So a task was formulated and added to an agenda, and eventually it was worked on. That task said to try to form generalizations of *H*12. Many heuristics applied (were potentially relevant to satisfying this new task), and created such new units. One heuristic noticed that the main connective in the If-Potentially-Relevant slot of *H*12 was AND, and decided to generalize *H*12 by replacing that connective with OR. Indeed, that new heuristic *H*12′ *did* claim it was potentially relevant much more often, but it never was *truly* relevant any more often, nor did it take noticably less time to evaluate H12's If-Potentially-Relevant slot. All in all, this kind of generalization had turned out to be a mistake. When EURISKO detected this, it eliminated *H*12′, and it synthesized a few new heuristics each of which—if they had only existed earlier—would have prevented *H*12′ from ever being allowed to survive. One

of these said to never replace AND by OR in an If slot of a heuristic; one said never to generalize the If Potentially-Relevant slot of a heuristic; one said never change the main connective in a slot. Since EURISKO knew that most of these heuristics would be wrong, or at least extreme, it gave each one only some chance of being followed, and detailed records were kept of their performances. Ultimately, the first new heuristic (and a variant of the second one) remained as permanent entries in EURISKO's knowledge base.

It is important to note the level at which EURISKO is working: it finds new concepts and conjectures in, say, naval fleet design. It finds new heuristics in that domain as well, as it also finds some new heuristics about how to find new heuristics. Strange 'bugs' can arise at those two highest levels; we give an example of each that EURISKO encountered:

One of the first heuristics that EURISKO synthesized (*H59*) quickly attained nearly the highest Worth possible (999). Quite excitedly, we examined it and could not understand at first what it was doing that was so terrific. We monitored it carefully, and finally realized how it worked: whenever a new conjecture was made with high worth, this rule put its own name down as one of the discoverers! It turned out to be particularly difficult to prevent this generic type of finessing of EURISKO's evaluation mechanism. Since the rules had full access to EURISKO's code, they would have access to any safeguards we might try to implement. We finally opted for having a small 'meta-level' of protected code that the rest of the system could not modify.

The second 'bug' is even stranger. A heuristic arose which (as part of a daring but ill-advised experiment EURISKO was conducting) said that all machine-synthesized heuristics were terrible and should be eliminated. Luckily, EURISKO chose this very heuristic as one of the first to eliminate, and the problem solved itself.

(5) *Representation.* The fifth task domain not discussed earlier is that of representation of knowledge. This is a very difficult area, one in which people have not made dramatic inroads in the last few millenia. EURISKO's task is quite constrained, actually: look for useful new slots which are specific to the various domains you are working in. Some heuristics guide EURISKO in deciding when it's time to define a new type of slot. For instance:

> **If** the average number of entries on *s* slots (for those units that have *any* entries on an *s* slot) is quite high—i.e., three times the average over all types of slots—
> **then** try to find specializations *s*; that will enable the partitioning of entries on all the *s* slots in the system.

When the VLSI domain was explored, one slot that became overtaxed was Parts. Previously, it had had very few entries, but now devices were coming along with dozens and sometimes hundreds of parts. The rule above fired, and a task was formed, to try to specialize Parts. When this task was ultimately

chosen and worked on, other heuristics guided EURISKO into meaningful splittings to make:

If you must decide how to specialize slot *s*,
then try to find a set of predicates that cover the entries;
some suggestions are: defined the same way, same syntactic type, used same way, related to same Topic, structurally similar.

In the case of the Parts slot, this heuristic succeeded in causing a split based on syntactic type: gates, doped regions, channel regions, wires, etc. had separate recognizable formats. Henceforth, each VLSI device had no explicit Parts slot; rather, it had slots called Gates, DopedRegions, etc. Later, when the Terminals slot was being strained to its limit, EURISKO suggested splitting it into InputTerminals and XorOutputTerminals (sets of terminals having the property that precisely one element in each set can be an output terminal).

Once a new slot is defined, existing heuristics can readily be specialized to deal with it (i.e., those that already deal with generalizations of that slot). Each slot has a definition, remember, and existing units can have this slot (according to its MakesSenseFor slot) will have some of their existing slots shortened or replaced, by adding this new slot and some of the entries that used to exist elsewhere on the unit. Incrementally, the slot is integrated into the network of slots that defines the representation of knowledge in the system.

This type of activity—formulating new domain-specific slots—happened rarely, but we believe it to be one of the most important long-range activities EURISKO can do. When a vocabulary is well chosen, thoughts become easy to express; the set of slots is essentially such a vocabulary, and must be augmented as new domains are explored.

4. Conclusions about Mechanizing the Process of Discovery

From the work on AM and EURISKO, we have acquired some insights into automated discovery. Some domains are better suited to this process than others, and the discovery program must contain certain elements in any case. We summarize these conclusions here, and then use them to explain the following phenomena: why AM worked so well, why AM ultimately failed, why it took so long to do EURISKO, and why EURISKO now works. Finally, we use the conclusions to explain our plans for future research in this area.

(1) *The domain should be as little explored as possible.* Fields which are already very well understood are not promising candidates in which to search for new discoveries, either at the domain- or at the heuristic-level. Until a machine can match human breadth of vision, insight, sources for metaphor and analogy, etc., its main advantage must lie in breaking new ground rather than scouring old ground for neglected gems. Set theory is not likely to yield many

new results easily; young fields such as graph theory are more promising; neonatal fields such as VLSI design are even more attractive. Another danger in choosing a very well-established field is that by now much of even the *research* activities in that field have become algorithmic and scripted.

But what if one's goal is to explore the phenomenon of discovery, not particulary to produce useful new ones? We still advise steering clear of well-developed fields. In them, not only has knowledge accreted, but an adequate representation has also formed in which to hold that knowledge. If your program starts with this adequate representation, which it is likely to do, then the discovery of facts and heuristics will be greatly facilitated—which will give you misleading information about the process of discovery. For example, even though AM later discovered arithmetic, little significance can be attached to that, as it already possessed the notion of bag (multiset), which is the natural way to represent arguments to arithmetic functions.

(2) *There must be a way to simulate—or directly carry out—experiments.* The field of exobiology satisfies the former criterion of being almost completely unexplored, but fails miserably on this one. Building a program to suggest experiments in molecular genetics sounds like a promising task—until one asks how the proposed experiments will be evaluated, how the program is supposed to evaluate partially worked out hypotheses along the way, etc. In some cases the outside world can be replaced by a teletype hookup to a human expert, but this is never quite as good as working in a field which is represented internally in the machine. Two approaches to this are a formalization (such as the axioms and definitions for some field of mathematics) and a simulation (such as a set of routines that compute answers to Mechanics situations they are asked about). One disadvantage of simulators is that the discovery program cannot go 'beyond' a reformulation of the same knowledge that went into the simulator. Given a Newtonian simulator, the program may come up with Newton's laws, or Lagrange's, but not Einstein's.

(3) *The 'search space' should be too immense for other methods to work.* No human should be able to manually, exhaustively search the same space as the program is walking around in. One big advantage the machine has over the person is that of tirelessness. It is not an accident that EURISKO's searches in the space of fleet designs consumed over a thousand CPU hours, nor that its VLSI explorations took ten times that much.

Usually this criterion means "too big for systematic exploration", but all we are requiring is that it mean "too big for manual exhaustive search". DENDRAL's problem space is a good example: chemists claimed (in refereed articles published in the best chemical journals) to have found all structural isomers of various formulae; DENDRAL was able to systematically search the space, and often found omissions from these lists. This is a case of problem solving rather than discovery, but the constraint we are trying to articulate is a general one of

what it means for a problem to have the right 'size' for attack by AI methods.

Note that this criterion has a converse: the search space should not be too immense for heuristic methods to work. Looking for useful VLSI devices is an example of this bracketing; interesting devices arise about once in 500 plausible candidates examined. If they were once in a billion, the task would not be suitable for automation in the 1980s; if they were once in five, one wouldn't need an AI program to find them.

(4) *There should be many objects, operators, kinds of objects, and kinds of operators. They should be related hierarchically and in other ways.* This makes a frame-based (class-oriented) representation useful: Each "way in which two entities can be related" is a different type of slot. A hierarchical organization makes the usual modes of inheritance important. The large number of objects and operators raises the need for an intelligent program, one which can keep track of complex interactions among many entities.

(5) *The task domain must be rich in heuristic structure.* Complexity of the domain raises the utility of plausible, inexact reasoning, as more precise inference becomes unmanageable or impossible. There should be many good heuristics which can apply, and no good algorithms. Theorem proving in propositional calculus is a poor domain for automated discovery, as it admits only a few heuristics; it is an even worse domain for discovery of new heuristics, because what few heuristics *do* apply in propositional calculus are already well known. Let us be a little more specific about the need for heuristics vis-a-vis algorithms:

(6) *There must be ways to generate, to prune, and to evaluate.* Many heuristics of each of three types should be available: heuristics which *generate* (suggest plausible moves), heuristics which *evaluate* (judge the worth and specific problems with the discoveries), and heuristics which *prune* (eliminate implausible paths before they are explored too deeply). It is acceptable for an algorithm to exist for one or even two of these processes, but not for all three.

(7) *The 'language' one uses to represent the concepts must be a natural one, given the set of objects and operators.* This is one of the most crucial conclusions, and one we did not arrive at until recently. At a very abstract level, one can view the domain task as being one in which domain operators (e.g., lab procedures, mathematical functions) are applied to domain objects (e.g., cultures, sets). In addition, EURISKO has a collection of higher level operators which combine domain entities (both objects and operators) into new ones (e.g., AddPlasmid, Compose, Conjoin) and which perform surgery on individual domain entities to produce modified ones (e.g., Mutate, Generalize, Coalesce). The task of an AM-like discoverer is to apply these higher-level operators in a fruitful, efficient manner, having a high 'hit rate'. That is, a high percentage of the time the result should be a new, useful domain entity. The

task of a EURISKO-like discoverer is to find heuristics which guide the application of the higher-level operators, so that the results *will* often be fruitful. If the high-level operators are well matched to the way the domain entities are represented, then not too much guidance will be required; most of the applications *will* yield meaningful new concepts. If there is a serious mismatch, then the problem may not be remedied even by a good set of guiding heuristics. See [1].

An example of a mismatch is one of differing granularity: the high-level operators are good at working on 2-line chunks of 'code', but your representation only includes three types of slots, so each unit has three enormous chunks of code that 'represent' it. When a mismatch is present, either the high-level operators or the details of the representation scheme must be adjusted until a match is (re)established.

But the high-level operators are nearly domain-independent: compose, coalesce, repeat, disjoin, weaken, etc. So almost all the necessary accomodation must be done not by them but by the representation scheme in which the domain knowledge is encoded. For instance, if a frame-based representation is employed, then the set of slots must be adjusted until the right 'granularity' is achieved (e.g., each slot having about two lines worth of entries).

In other words, even though the discovery of new heuristics is important, the presence (and maintenance) of an appropriate representation for knowledge is even more necessary. Once you do have such a match, as in AM's case, the main problem then appears to be the discovery of new heuristics.

(8) *Criteria which make a domain suitable for* AM*-like exploration* (*discovery of new concepts and conjectures*) *are—taken to extremes—the same criteria which make a domain suitable for* EURISKO*-like exploration* (*discovery of new heuristics*). This is an interesting corollary to the need for heuristic structure. To be well-suited to AM-like exploration, a domain must be open-ended, uncharted, internally formalizable, and possess a rich structure of heuristics. In extreme cases, the domain is so unexplored that not even the heuristics are available; i.e., there are no human experts in the field. In that situation, EURISKO's approach may be fruitfully applied, since *any* useful heuristics it produces will be welcome new discoveries.

5. Interpreting AM and EURISKO in Light of These Conclusions

Of the domains in which EURISKO has so far been applied, the two which most closely satisfy all the above criteria are the Traveller TCS game and the three-dimensional VLSI design task. These are in fact the two areas in which EURISKO has discovered valuable domain-level concepts and useful new heuristics as well.

In this subsection we consider the behavior of AM and EURISKO, applying the

previous criteria to explain successes, failures, and difficulties encountered. This is an admittedly circular argument, since those criteria were abstracted from just such experiences. Only future research with EURISKO, in new task domains and more deeply in its present ones, will be able to test those claims.

First, we address the issue of why AM worked. The denseness of useful mathematical concepts appears crucial; namely, that a large fraction of the time, when we modified some old concepts, the things we got were useful new concepts. As the next to last criterion above indicated, denseness is dependent on the set of operators one has for getting new concepts, and the representation one uses. In AM, the representation was frame-based at a superficial level, but each concept's definition was a single chunk of LISP code. That is, each concept was supplied with a LISP program which computed its characteristic function. To see if X is a SetOfSets, e.g., one goes to the concept called SetOfSets, look for its Defn property, and finds an expression like

```
(λ (s) (AND (Apply* (Defn Set) s)
            (EVERY s (Defn Set))))
```

This predicate checks that s is a set and that so is every element of s. What AM did, typically, was to modify such characteristic functions, combine them, etc., and then—once it had a new piece of LISP code—see what concept it was the characteristic function of. For instance, if asked to generalize SetOfSets, AM could substitute List for Set, and get a new piece of LISP that said

```
(λ (s) (AND (Apply* (Defn List) s)
            (EVERY s (Defn List))))
```

AM would then simply assume that this was the characteristic function for *some* concept similar to, but more general than, SetOfSets. It would set up a new unit, giving it this predicate as a Defn, and eventually might get around to trying to find examples of such things, look for conjectures about them, and so on. Often, as in this case, the result was indeed meaningful.

Thus AM was actually not walking around in the space of mathematical concepts, it was walking around in the space of 'small LISP predicates'. Its primitives were functions that modified LISP predicates, combined them, etc., and the reason AM achieved good results is because these high level operations, applied to short LISP code for characteristic functions, often yields short LISP code for characteristic functions of different but useful concepts. When cast in this form, it appears much more like a fluke that AM worked. It is thanks to the natural relationship between LISP and mathematics (therefore thanks to John McCarthy, Alonzo Church, and others) that common math functions can be stated so succinctly in LISP. Brevity is a key attribute in any kind of asemantic exploration. If useful concepts are short expressions in your language, then you have some chance of coming across them often, even if you don't know much about the terrain.

As AM worked on, it built up larger and larger definitions for its derived concepts; instead of being a couple lines long, they became half a page in length. The old high-level combiners and mutators no longer were able to maintain a high 'hit rate'. AM needed at least one of the following:

(i) new high-level operators (never likely to happen too often!),

(ii) new heuristics to guide the process so that syntactic serendipity would not have to be relied upon (this is what EURISKO was aimed at),

(iii) a new and different set of slots (or, equivalently, a different set of programming primitives than LISP), so that the concepts' definitions would once again be short expressions which the high-level operators could work on fruitfully, or, finally,

(iv) a good interface to a human expert, so as to work in the mode known as man–machine interaction [2].

The final two alternatives are the most powerful and plausible. The fourth one, man–machine interaction, is not so useful in the fields EURISKO explores, as there *are* as yet no human experts in Traveller TCS, 3D VLSI design, etc. The third alternative was not considered until 1979. In 1976 we began trying to simply get a program, like AM, to get new heuristics. The obvious approach was to let the heuristics (which served AM so well for so long) apply to each other. Time after time, the results were terrible. We now see the reason: each heuristic, though represented superficially as a unit, had two executable slots that basically defined it: IF and THEN. Each of those slots had a large chunk of LISP code in it, and the heuristics tried valiantly to guide high-level operators as they combined and modified these huge chunks of LISP code. The situation here was one of an even worse mismatch than existed in AM when we gave up on that; many of the heuristics had IF or THEN slots that were over a page long. This problem, and its solution, had been remarkably well predicted by Amarel [1] many years earlier.

Gradually, over the past six years, our attempts have met with more and more success. What had we been doing? As time went on, we found ourselves defining more and more kinds of slots that a heuristic might have, and occasionally new types of slots for 'object-level' concepts as well. This new language allowed the size of the pieces of LISP code on each slot of each heuristic to shrink. As the average size declined, from 60 lines to 3, the old high-level operators (combiners and mutators) began to produce a high percentage of 'winners' once again. Valid, valuable heuristics were being synthesized. To prevent EURISKO from eventually thrashing, this set of slots must be dynamically expandable, and indeed that has been a major recent focus of our work. For every seven heuristics EURISKO finds, on the average, a new kind of slot is defined.

Once one tackles the problem, it is not difficult to find a useful set of slots to replace IF and THEN. The method we (and EURISKO) use is to look over the current IF and THEN slots' values, looking for commonality in the code

therein. For instance, many rules called PRINT near the very end of their execution; that caused us to add ThenFinallyPrint as a slot that heuristics could have, give it the proper definitions, eliminate the print commands from the THEN slot of all the heuristics, and add them (in shorter form, since we could leave off the common details) to the ThenFinallyPrint slots. This happened over and over again, for various categories of tests and actions, until now we no longer have an IF or THEN slot per se. Each type of test or action falls into the purview of some specialized kind of slot. If a proper kind cannot be found, that is a signal (to us, and now to EURISKO) that a new kind of slot should be defined.

The criteria in Section 4 are guiding our present research directions on EURISKO. We are focussing on domains which are large, unexplored, complex, and rich with heuristic structure, and being very conscious to employ a representation which is well matched with our set of high-level concept-synthesis operators. The task which satisfies these criteria most closely is the design of three dimensional VLSI devices, and that is the task we are choosing to concentrate upon. More theoretically, we are investigating ways to discover appropriate new slots, judgmental rules for monitoring the goodness of match between representation and high level operators, and new entries for the set of high-level operators that generate, prune, and evaluate new concepts.

Our original 1976 assumption was that heuristics could be treated just like math concepts, and we could apply the same methods (heuristic search) to discover new ones. But we were fortunate in choosing elementary mathematics as the test domain for AM; heuristics *are* like most other domains, it's *mathematics* that's special and (thanks to LISP) particularly easy. As demonstrated by the recent successful performances of our EURISKO program, we are developing an understanding of what it takes to find new concepts in other fields, including the discovery of new heuristics.

ACKNOWLEDGMENT

This work has benefited from the many useful comments by Saul Amarel, Dan Bobrow, John Seely Brown, Bruce Buchanan, Ed Feigenbaum, Greg Harris, Judea Pearl, Elaine Rich, James Saxe, Mark Stefik, and the late John Gaschnig. EURISKO is written in RLL, a self-describing and self-modifying representation language constructed by Russ Greiner, Greg Harris, and the author. I am grateful to XEROX PARC's CIS and Stanford University's HPP for providing unparalleled computational environments, and to ONR (N00014-80-C-0609), ARPA, and XEROX for financial support.

REFERENCES

1. Amarel, S., On the representation of problems of reasoning about actions, in: D. Michie (Ed.), *Machine Intelligence* 3 (Elsevier, New York, 1968) 131–171.
2. Bledsoe, W.W. and Bruell P., A man-machine theorem-proving system, *Artificial Intelligence* 5 (1974) 51–72.
3. Davis, R. and Lenat, D., *Knowledge Based Systems in AI* (McGraw-Hill, New York, 1981).

4. Feigenbaum, E.A., Knowledge engineering: the practical side of artificial intelligence, HPP Memo, Stanford University, Stanford, CA, 1980.
5. Gibbons, J. and Lee, K.F., One-gate-wide CMOS inverter on laser-recrystalized polysilicon, *IEEE Electron Device Letters* **1**(6) (1980).
6. Hayes-Roth, F., Waterman, D. and Lenat, D. (Eds.), *Building Expert Systems, Proc. 1980 San Diego Workshop in Expert Systems* (Addison-Wesley, Reading, MA, 1982).
7. Knuth, D., *Surreal Numbers* (Addison-Wesley, Reading, MA, 1974).
8. Lenat, D.B. and Greiner, R.D., RLL: a representation language language, *Proc. First Annual Meeting of the American Association for Artificial Intelligence* (AAAI), Stanford, CA, 1980.
9. Lenat, D.B., The nature of heuristics, *Artificial Intelligence* **19**(2) (1982) 189–249.
10. Lenat, D.B., Learning by discovery: three case studies in natural and artificial learning systems, in: R.S. Michalski, T. Mitchell and J.G. Carbonell (Eds.), *Machine Learning* (Tioga Press, Palo Alto, CA, 1982).
11. Lenat, D.B., Sutherland W.R. and Gibbons, J., Heuristic search for new microcircuit structures, *AI Magazine* 3(3) (1982) 17–33.
12. Lenat, D.B., Theory formation by heuristic search; the nature of heuristics II: Background and examples, *Artificial Intelligence* **21**(1, 2) (1983) 31–59.
13. McCarthy, J. and Hayes, P., Some philosophical problems from the standpoint of artificial intelligence, in: B. Meltzer and D. Michie (Eds.), *Machine Intelligence* **4** (Edinburgh University Press, Edinburgh, 1969) 463–502.
14. Wiseman, Results of the 1981 Trillion credit squadron competition, *J. Travellers Aid Soc.* (1981).

Received August 1982; revised version received October 1982