AUTOMATED DISCOVERY

by

Paul R. VanArragon

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

Research Report CS-85-14
May 1985

Automated Discovery

by

Paul R. Van Arragon

An essay
presented to the University of Waterloo
in partial fulfillment of the
requirements for the degree of
Master of Mathematics
in
Computer Science

Supervisors:
Marlene Jones Colbourn
David L. Poole

Waterloo, Ontario, December, 1984

Acknowledgements

Special thanks to Marlene Colbourn, who not only officially supervised this project, but also suggested the topic, located references, guided my thinking, provided encouragement, taught me how to write, and even offered financial support from her swiftly-depleting research grant. (What else could she have done?) Thanks to David Poole, who also guided my thinking and revealed errors in earlier drafts of the essay. Thanks to the department of Computer Science for further financial support.

Finally, thanks to my roommates, Gord Alton and Bruce Meyers, who, although they have not studied the problem discussed herein and did not consciously assist me, by often being present in the same room as me while I worked on the project, may have unwittingly helped in some subtle, passive and vicarious way.

Paul R. Van Arragon
December 11, 1984

Automated Discovery

Paul Van Arragon

University of Waterloo

ABSTRACT

Discovery learning is an area of Artificial Intelligence in which computer scientists attempt to automate the process of scientific discovery. We discuss the definition of discovery learning, and review the previous work done in the area, including Lenat's AM (that researches mathematics), EURISKO (a revision of AM that studies several domains), Langley's BACON (that discovers empirical laws), and Aref's system (that discovers data structure concepts). We critique each program and suggest directions for future research.

Table of Contents

Automated Discovery

Paul Van Arragon

University of Waterloo

1. Why Automate Discovery?

Imagine a computer program that is able to conduct scientific discovery. Such a program is of interest to artificial intelligence (AI) researchers who study learning. It could be used by scientists to uncover new knowledge or by psychologists to study the discovery process. Discovery programs have been built for many domains, including mathematics, game playing, physical laws, and data structures.

Below is a session with a fictitious discovery program. The program contains knowledge about data structures, mathematics and logic; it's task is to discover methods of sorting lists of numbers. It knows nothing explicit about sorting, but has heuristics and production rules that can synthesize algorithms. (Comments in the following session are in italics.)

User: Discover sorting algorithms; that is, programs that, given a list of numbers, output the list in sorted order.

(Assume that the computer understands the intended meaning. There may be a lengthy pause before the computer answers.)

Computer: Success. The following algorithm sorts:

```
Sort(A[1..N])
  COMMENT: output a sorted array B[1..N]

  For i:= 1 to N
     begin
```

```
          minimum := A[1]
          position := 1
          For j := 1 to N
             begin {Find smallest element of A.}
                If minimum > A[j]
                then minimum := A[j]
                   position := j
             end

          {Remove the smallest element.}
          A[position] := infinity
          B[i] := minimum
       end
```

(This is an inefficient version of selection sort.)

User: Give me information about its complexity based on comparisons where N is the length of the list being sorted.

```
Computer:  Worst Case Time:  O(N^2)
            Best Case Time:  O(N^2)
            Average Time:    O(N^2)
```

User: Can you find a lower bound on sorting algorithms?

Computer: Success. Lower bound: O(N)
Proof: All numbers have to be compared at least once.
There are N numbers.

User: Can you find a tighter bound?

Computer: Timed Out. I could not prove or disprove:
'A tighter bound for sorting exists.'
Do I have more time?

User: No. Try to find another Sorting algorithm with lower complexity. Use this heuristic: 'Organize the data so that more information is kept about previous comparisons.'

Computer: Another algorithm found:

(Here the computer outputs the algorithm for bubble sort. We omit it to save space.)

Best Case Time: O(N^2)
Worst Case Time: O(N^2)
Average Time: O(N^2)

Still searching.

Another algorithm found.

(A variation of bubblesort in which the algorithm stops passing through data when it notices that no exchanges have been made is output here.)

Best Case Time: O(N)
Worst Case Time: O(N^2)
Average Time: O(N^2)

User: Can you find an algorithm that sorts in average time O($N\log N$)?

Computer: Success.

(The Heapsort algorithm is output here.)

Best Case Time: O($N\log N$)
Worst Case Time: O($N\log N$)
Average Time: O($N\log N$)

User: Try again to find a tighter lower bound for sorting.

Computer: Success. O($N\log N$) is a lower bound for the average case time for sorting a list of length N.

(The proof follows.)

The above session demonstrates the kind of system that some researchers would like to build. Moreover, there is interest in developing software to discover genuinely new things instead of the above well-known sorting algorithms. The utility of such a program as a research tool is obvious. Because of its speed, a computer can search the space of possible discoveries in certain domains faster than a person. If the software contains enough expert knowledge to prune the search space, it might find important discoveries.

Lenat, who has done extensive research in the area [6, 7, 9, 11, 14, 16, 24-30, 38], explained why discovery programs would be useful:

> Why should AI be concerned with computer programs which learn by discovery? One obvious answer is AI's interest in the mechanization of any human cognitive activity. There is another, more powerful reason, however. The standard approach to expert system-building involves extracting knowledge from human experts, and yet many of the young, explosively-growing, important fields have no human experts yet, and have few rules of thumb for guiding explorations in them. In such virgin territory discovery programs may be the fastest route to gaining a preliminary understanding, to conceptualizing the useful new objects, operations, and heuristics of those fields. [14]

The above idea is enticing. One interesting application of it is in the field of mathematics. We understand a formal system of mathematics--the axioms and the rules of inference--hence, we could indefinitely generate provable theorems, and thereby discover. But, unless the generation is guided, many useless theorems will result; we must provide the computer with appropriate heuristics to prune fruitless productions from the search space.

Developing heuristics is difficult. Unfortunately, even prominent mathematicians cannot say what heuristics should be employed. Many mathematicians and scientists have thought about automating discovery, or have tried to understand the process, but all had difficulty discerning the heuristics used.

Poincaré [15, 31], tried to explain mathematical discovery:

> What is mathematical creation? It does not consist in making new combinations with mathematical entities already known. Any one could do that, but the combinations so made would be infinite in number and most of them absolutely without interest. To create consists precisely in not making useless combinations and in making those which are useful and which are only a small minority. Invention is discernment, choice. [31]

Poincaré performed mathematical creation himself, but failed to isolate his heuristics because, he claimed, his subconscious seemed to be doing much of the work. He wrote of an experience he had while trying to prove that there could not be any functions like those he called Fuchsian functions [21]. He spent 15 days unsuccessfully trying to prove the desired result; later, while boarding a bus, a proof came to him while he was not consciously thinking about it. Such experiences led him to believe that his subconscious was able to apply heuristics automatically. In his own words:

> The unconscious, or, as we say, the subliminal self plays an important role in mathematical creation; this follows from what we have said. But usually the subliminal self is considered as purely automatic. Now we have seen that mathematical work is not simply mechanical, that it could not be done by a machine, however perfect. It is not merely a question of applying rules, of making the most combinations possible according to certain fixed laws. The combinations so obtained would be exceedingly numerous, useless and cumbersome. The true work of the inventor consists in choosing among these combinations so as to eliminate the useless ones or rather to avoid the trouble of making them, and the rules which must guide this choice are extremely fine and delicate. It is almost impossible to state them precisely; they are felt rather than formulated. Under these conditions, how imagine a sieve capable of applying them mechanically? [21]

Polya [32, 33], was more helpful. In the following text, he mentions several heuristics that can be employed in problem solving:

> Trying to solve a problem, we consider different aspects of it in turn, we roll it over and over incessantly in our mind; VARIATION OF THE PROBLEM is essential to our work. We may vary the problem by DECOMPOSING AND RECOMBINING its elements, or by going back to the DEFINITION of certain of its terms, or we may use the great resources of GENERALIZATION, SPECIALIZATION, and ANALOGY. Variation of the problem may lead us to AUXILIARY ELEMENTS, or to the discovery of a more accessible AUXILIARY PROBLEM. [32]

Like Poincaré, Polya concluded that much of the work is done by the subconscious; hence, one of his heuristics was *take counsel of your pillow*. Furthermore, he claimed that infallible rules of discovery are beyond the scope of serious research. He referred to Descartes, Leibnitz, Bolzano, Pappus, and Hadamard; all of whom had tried to explain the thought processes used in problem solving and discovery [32]. He wrote:

> To find unfailing rules applicable to all sorts of problems is an old philosophical dream; but this dream will never be more than a dream. [32]

Despite its apparent difficulty, computer scientists have attempted to tackle the problem. As mentioned earlier, several discovery programs have been built. Simon commented on these programs and their inherent view of discovery:

> The theory of discovery that emerges from an examination of how these [discovery] programs work contains little that should surprise us; unless we have been seduced by the often-repeated myth that discovery processes, being 'creative', somehow stand apart from the other actions of the human mind. [37]

Lenat echoed his sentiments and suggested how to solve the problem:

> 'How was X discovered?' When confronted with such a question, the philosopher or scientist will often retreat behind the mystique of the all-seeing I's: Illumination, Intuition, and Incubation. A different approach would be to provide a rationalization, a scenario in which a researcher proceeds reasonably from one step to the next, and ultimately synthesizes the discovery X. In order for the scenario to be

> convincing, each step the researcher takes must be justified as a plausible one. Such justifications are provided by citing heuristics, more or less general rules of thumb, judgmental guides to what is and is not an appropriate action in some situation. [27]

Indeed the problem is hard, but by finding proper heuristics, progress can be made. The systems discussed in the following chapters provide a deeper understanding of the problem. By improving on their weak areas and adopting their strong points, better discovery systems can be developed.

2. Toward a Definition

Like many technical terms of AI, discovery learning is hard to define. It is important to have a definition to provide a framework for discussion. In addition, a definition can state goals that a discovery program ought to attain, thus giving us grounds for evaluation. Without getting entangled in details, we look for an adequate working definition: one that does not rely on the internal workings of a program and that correctly isolates the class of programs we wish to discuss. We consider several definitions and choose one that satisfies our intuitive notion.

2.1. A Definition of Learning

Discovery is usually considered to be a type of learning. Therefore, as a basis for our definition of discovery, we first choose a definition of learning, then we decide what distinguishes discovery from learning in general.

Unfortunately, learning has no commonly accepted definition. Most people agree that learning is accretion of knowledge, but this definition is too broad when we apply it to computer programs. Even payroll programs accrete knowledge in some sense; they gain knowledge about employee payment records, but they are not called learning programs.

The Handbook of AI [8] gives four perspectives on learning: any process by which a system improves its performance, the acquisition of explicit knowledge, skill acquisition, and theory formation. We consider each of these four perspectives as definitions.

To help criticize these definitions, we first describe briefly four discovery learning programs.

1. BACON discovers empirical laws. It performs a heuristic-guided search to find a rule that explains experimental data. For example, BACON discovered the ideal gas law: PV=NrT. Measurements from several experiments were given as input. BACON looked for constancies and found that PV=NrT was true for all the given data.

2. AM discovers elementary mathematical concepts. It starts with a wide variety of fundamental concepts such as lists and sets, and many heuristics that combine and manipulate the concepts to discover new mathematics. AM defined the concept *prime numbers* by specializing *numbers* to produce *numbers with exactly 2 divisors.*

3. EURISKO is a descendant of AM. It is able to work in different domains, and has meta-heuristics for discovering new heuristics. One of its domains is naval fleet design. The program, referred to as EURISKO/Navy, designs fleets of ships based on the rules of a game. Battles are simulated and the results are used to improve the fleet.

4. Another EURISKO domain is Very Large Scale Integration (VLSI) circuit design. We refer to the program as EURISKO/VLSI. It searches through the space of possible VLSI designs, with heuristic guidance, to discover useful designs.

The first definition, *learning is any process by which a system improves its performance,* assumes that the program is performing a task. The definition excludes programs that learn but do not improve on a task. BACON is such a program; it learns about constancies in data, but does not improve as it runs. It is unable to find constancies better when it halts than when it begins. It learns about the constancies, the empirical laws; it is not improving on how it finds them. Moreover, AM falls into the same category: it learns mathematics but does not use them in a performance task.

Perhaps a better definition is that *learning is theory formation.* BACON is performing theory formation in the sense that it forms theories about consistencies in the real world based on the data it receives. The theories explain the data and can be used to predict results of future experiments.

AM also forms theories. It conjectures relationships concerning the concepts that it knows. For example, it conjectured that 2 is the only even prime. Even EURISKO/VLSI forms a theory; as it finds new designs, it develops a theory of how to find designs faster.

So far theory formation seems to be a suitable definition. In the case of EURISKO/VLSI, however, we prefer to use another definition. The program has two goals; not only does it produce a theory of how to design VLSI circuits, it is actually designing circuits, or learning new circuit designs. Its ultimate goal is to learn new designs, and it can be included as a learning program on this basis alone.

EURISKO/Navy causes a similar problem. It forms a theory of fleet design, but only as a secondary aspect of its learning. By discovering fleet designs it learns to play the game better. Hence, the previous definition, that *learning is a process by which a system improves its performance,* applies here: EURISKO/Navy is improving its performance at the battle game.

Perhaps the definition *learning is the acquisition of explicit knowledge* is preferable. By explicit knowledge, we mean information that is easily isolated from the control structure of the program. EURISKO/Navy gains explicit knowledge about fleets. EURISKO/VLSI gains explicit knowledge about VLSI designs and design techniques. BACON learns empirical laws and AM learns mathematical concepts, both explicitly. This definition's shortcoming is that it says little more than our original definition: *learning is the acquisition of knowledge.* The extra requirement, that the knowledge be explicit, may even damage the definition. Not only does it still include payroll programs (because they learn explicit knowledge about employee records) but it may also eliminate some programs that *are* considered to be learning. When humans learn, often their learning is implicit. As they mature, they are learning about 'life', yet their knowledge is partly inarticulatable. Likewise, a computer could learn implicitly. Although most learning programs do learn explicit knowledge, we should not restrict the definition to those only, because non-explicit knowledge acquisition seems possible.

The remaining definition is *learning is skill acquisition.* Generally, *skill* refers to a physical activity. In that sense, it does not apply to any of the programs described earlier; hence, it is inappropriate.

To avoid the problems of choosing a more specific definition, we adopt the more general definition *learning is the acquisition of knowledge* and apply any one of the perspectives as appropriate. For example, EURISKO/VLSI learns explicit knowledge: designs of circuits. It also forms a theory of how to design circuits, and it improves in the activity of designing. Hence, we can critique it in the context of all three perspectives. We should not limit our definition so that any of the perspectives is not called learning. For simplicity, we would like to narrow our definition, but the word 'learning' is too general to allow it. If we wish to discuss a more specific idea and to judge a program with regard to that idea, we choose a constraining perspective and include that as part of our definition. For example, we may want to speak of EURISKO/VLSI as a theory formation program because it is building a theory of how to design VLSI.

2.2. A Definition of Discovery Learning

Now the question is, what is unique about discovery learning? What kind of learning is it? We consider several possibilities.

One idea that seems to differentiate the two is that *discovery programs work internally without interaction with the outside world* whereas learning programs receive data from outside. Both AM and EURISKO, for example, are based on a formalism. Concepts are built and relationships between the concepts are found based on a formal system; no external interaction takes place.

Such a definition, however, is misguided. Most of real-life discovery involves interaction with the outside world. BACON mimics real-life discovery by discovering empirical laws via experimental data, therefore it does not satisfy the above definition. Discovery can happen in formal systems *and* in the real world via experiment.

Bruner [8], writes of learning by discovery as a technique for teaching. It involves withholding knowledge to force the student to discover things by experimentation. This implies that discovery is *learning without being told.*

Lenat uses a similar definition: *discovery programs are those that have been given less human guidance or forethought* [28]. He suggests that there is a continuum of how much a program is being told. Discovery programs fit on the continuum near the extreme of not being told anything. One problem with this definition is that it is difficult to rate a program on this basis. A common criticism of AM is that it has been told too much, thus its discoveries are trivial, but how can such a judgement be made? In the most trivial sense, AM has been told exactly what is necessary (the original concepts and heuristics) to produce its discoveries. At what level should we decide how much a program has been told?

When teaching a student discovery techniques, we may give him or her some of the information needed to solve a problem. The student has to find an answer without further help. We do not know what knowledge the student uses, but we say that he or she has learned by discovery, or has learned without being told. Here the judgement of how much has been told is easier to make. We simply consider only that which is told at the time; we neglect any knowledge the student has learned elsewhere. One way of deciding how much a program has been told is by the amount of inference required, but even this hardly helps to classify programs clearly. Because of this difficulty, we consider other definitions.

Simon defined discovery programs as *those whose output is not inferable in any obvious way from its input* [37]. This definition as it stands includes any non-understandable program or any program that generates random numbers, but Simon augments his definition by stressing that the output must be socially valuable or remarkable in some way. It cannot be something that is 'obvious to a person skilled in the art.'

This definition is also unsatisfying. How can we decide when a result is not obviously inferable? Every result is inferable in a sense, so again we are faced with a continuum; this time discovery programs are at the extreme of being the hardest to infer the result by hand. Like the previous definition, this one sheds some truth about discovery programs, but it is concerned with the inner workings and does not delineate clearly between discovery programs and others; therefore it is unappealing.

The dictionary definition of discovery is *to learn about something that is previously unknown*. If we mean 'unknown to mankind', the definition is too restrictive since most discovery programs rediscover knowledge. If by 'unknown' we mean 'unknown to the computer', it is not restrictive enough. We might say that a computer that multiplies two numbers discovers a product, and therefore is a discovery program. Again the definition holds some truth but is insufficient.

To solve (or to avoid) this problem, we use a more intuitive definition: *a discovery program is one that was designed to be able to learn new-to-mankind knowledge*. AM and BACON, discovery programs that only rediscover knowledge, are included in the definition because their intent is to discover genuinely new theories. Their success or failure should have no bearing on whether they are classified as discovery programs.

The benefit of this non-technical definition is that it isolates an important aspect of each system: the intent to make new discoveries. The ability to achieve this intent is the aspect of each discovery program we wish to discuss, so we accept this as our working definition. Other programs may include aspects of discovery, but we discuss only those that seem to have the overall intent of being a discovery tool.

2.3. Criteria for Judging Discovery Programs

As mentioned previously, the most common criticism of discovery programs is that they cheat in the sense that the heuristics chosen are precisely those that can produce the desired output. This criticism is hard to verify however because it involves the integrity of the designer of the program who may claim he used only general heuristics.

We attempt to judge in a more objective way. We assess the ability of the program to conduct research and to discover useful things. How we discuss a program depends on its domain. BACON is considered differently from AM since it discovers in the physical world, whereas AM discovers in a formal world.

In summary, a learning program accretes knowledge; a discovery program has the intent of learning new-to-mankind knowledge. We judge programs on their ability to discover genuinely new things.

3. AM

AM [6, 7, 10, 11, 24-27, 30, 35] discovers concepts and theorems of elementary mathematics. Its initial knowledge base contains 115 simple set theory concepts such as sets, lists, and equality, and 243 heuristic rules that produce new concepts and propose new theorems. The heuristic rules also suggest tasks that AM performs. Each task is given an interest value and is placed on an ordered queue called the agenda.

During each cycle of execution, AM performs the following loop:

```
take the most interesting task from the agenda
find any heuristics relevant to the task
perform the task and store the results (which may be new concepts,
    conjectures, and tasks)
```

AM was run several times. Each time about 200 new concepts were formed, 100 of which Lenat deemed interesting. It discovered ***numbers, prime numbers,*** and ultimately discovered ***the unique factorization theorem*** (every number can be uniquely factored into primes).

3.1. Concepts

A critical aspect of AM is its starting concepts; they are the building blocks on which AM makes discoveries. The ability to discover depends upon AM's initial concepts, and how they are represented. All of these concepts are prenumerical. They deal with sets, lists and logic; none concern numbers. They fall into two categories: activities and objects. Activities include logical relations (disjunction, conjunction, and implication) and operations (equality, invert and canonize). Objects include conjectures, identifiers, variables, lists and sets. Section 3.4 contains an example demonstrating these concepts, and their use.

Name: Set, Class, Collection.

Definitions:

  Origin: Created by Lenat.

  Recursive: lambda(S) [S={} or Set-Definition (Remove(Any-member(S),S))]

Examples: {{}}, {A}, {A,B}, {3}

  Boundary: {}, {A,B,{C,{{{A,C,(3,3,3,9),}}}}}

  Boundary-Failures: {A,A}, ()

  Failures: <4,1,A,1>

Generalizations: Unordered-structure, No-multiple-elements-structures

Specializations: Empty-set, Nonempty-set, Set-of-structures, Singleton

Conjectures: All unordered-structures are sets

Analogies: Bag, List, Ordered Set

Worth: 600

Figure 3.1: a starting concept frame: set

One may wonder why Lenat chose the concepts that he did. He explains that he considered choosing either a necessary or a sufficient set of concepts [14]. A necessary set includes all concepts that cannot be derived from the other concepts; a sufficient set is a set from which all of mathematics can be derived. Because of the difficulty of deciding which concepts belong to each of these sets, he decided to choose those concepts that are generally believed to be possessed by 4-year-old children. Hence, AM starts with no explicit knowledge of proof techniques, arithmetic, or infinity.

Each of the starting concepts is an interLISP frame with approximately 25 slots. As an example, the frame representing *set* is shown in figure 3.1. New discovered concepts are also LISP frames; the frame of a discovered concept, *prime number,* is shown below in figure 3.2.

Name: Prime numbers.
Definitions:
  Origin: Number-of-divisors-of (x) = 2
  Predicate Calculus: Prime(x) = ForAll z ($z|x$ -> z=1 or z=x)
  Iterative: (for x >1): For i from 2 to $\sqrt{x}$, i does not divide x
Examples: 2, 3, 5, 7, 11, 13, 17
  Boundary: 2, 3
  Boundary-Failures: 0, 1
  Failures: 12
Generalizations: Numbers, Numbers with an even number of divisors, Numbers with a prime number of divisors.
Specializations: Odd primes, Prime pairs
Conjectures: Unique factorization theorem, Goldbach's conjecture
Analogies: Maximally-divisible numbers are converse extremes of Number-of-divisors-of
  Factor a non-simple group into simple subgroups.
Interest: Conjectures associating primes with times and with divisors-of
Worth: 800

Figure 3.2: a created concept frame: prime number

Slots can be divided into three categories: those that relate the frame to other frames (Generalizations and Specializations), those that say something about the frame itself (Definitions and Worth), and subfacets that contain heuristics (not shown in the figures).

The Definition slot of a frame gives the concept its ultimate meaning. The LISP code therein defines the concept. The example slots contain structures and atoms that satisfy the definition. AM is able to instantiate definitions to find examples of concepts. Furthermore, it can divide examples into positive, negative, and even boundary cases. One method used is to take a boundary case from the base step of a recursive definition. For example, when finding examples of sets, it can take the set { } from the definition itself (see figure 3.1).

The generalizations, specializations, and analogies slots associate concepts with each other. These links enable AM to communicate discoveries between related concepts. When AM finds instances of a concept, it sends them to the more general concepts listed in the generalization slot. Heuristics are also shared in this way.

Plausible theorems concerning a concept are stored in its conjecture slot. AM finds these conjectures by noticing properties about the examples and non-examples. One heuristic conjectures that two concepts are equivalent if they share all the same examples.

The Interest and Worth facets contain information about the concept's importance. English sentences explaining this importance are stored in the interest slot. For instance, prime numbers were judged interesting partly because of the conjecture *all numbers with 3 divisors are perfect squares of prime numbers*. Every such discovered pattern contributes to the total worth of a concept.

Many facets of each frame have subfacets that contain heuristics. These are not shown in the diagram. Prior to running AM, Lenat decided where each heuristic could be used, and attached them to the relevant facets.

3.2. Heuristics

Heuristics are generally employed as pruners of a legal search space--rules of thumb that decide which of many possible paths to take during a search. AM's heuristics, however, define the search space. They are plausible space generators, instead of legal space pruners, because each heuristic defines a plausible path towards interesting discoveries.

The heuristics can be roughly divided into three types: those that fill in some slot of a concept, those that create new concepts, and those that add new tasks. These three roles are often combined: figure 3.3 shows a heuristic that fills in a slot (the conjecture slot) and also creates a task. The heuristic of figure 3.4 creates a new concept.

Heuristics are coded in LISP; those in the figures are written in English for readability. Each heuristic has two parts: an if-part and a then-part. The if-part has no side effects; it is evaluated whenever there is a possibility that the heuristic can perform the current task. If the if-condition returns true, the

```
If the current task is to Check Examples of any concept X,
    and (for some Y) Y is a generalization of X,
    and Y has at least 10 examples,
    and all examples of Y are also examples of X,

Then print the conjecture: 'X is no more specialized than Y'
    and add it to the Examples facet of the concept named Conjectures,
    and add the following task to the agenda: 'Check examples of Y',
    for the reason: 'Just as Y was no more general than X, one-of
        Generalizations(Y) may turn out to be no more general than Y,'
    with a rating for the reason computed as the average of
        ||(Examples(Generalizations(Y))||, ||Examples(Y)||, and
        Priority(Current task).
```

Figure 3.3: a heuristic: notice similar concepts

```
If a newly-interesting operation F(x,y) takes a pair of N's as arguments,

Then create a new concept, a specialization of F,
    called F-itself,
    taking just one N as argument,
    defined as F(x,x),
    with initial worth(F)
```

Figure 3.4: a heuristic: coalesce

then-part is executed and the task is accomplished.

An interesting aspect of the heuristics is how they form conjectures. In general, AM studies the examples facets of a concept to find unexpected patterns and equivalences. It conjectured that 2 is the only even prime by comparing *prime numbers* with *odd prime numbers* and discovering that the boundary case, 2, was the only variant. Based on 10 examples that were shared by both concepts, the conjecture was made.

3.3. The Agenda

The agenda is a simple mechanism that orders tasks by their interest value; it facilitates a best first search by telling AM which task is the most promising at a given time.

Each task on the agenda carries reasons for its importance. These reasons are supplied by the heuristic that created it and by other heuristics that notice interesting properties of concepts. Based on the numeric value of its supporting reasons, a task is inserted into the agenda. Since the reasons are in English, they can be given to the user to explain why a task was chosen.

3.4. An Example Discovery

The most lauded of AM's discoveries is *the unique factorization theorem*. Since AM has no theorem proving ability, it could not prove the theorem, but based on empirical study, the conjecture was made. The following is a summary of the path to that discovery.

1. The operation *set-equal* is generalized to discover an operation that tests whether two sets are of the same size (contain the same number of atoms).
2. The structure *set* is canonized into the concept *set containing T's*. AM looked for a predicate P such that if X and Y are sets, X and Y are the same size if and only if P(X) and P(Y) are equivalent sets. Sets of T's are analogous to numbers, so the concept was renamed *number* by Lenat.
3. The operation *append* is extended so that its domain is numbers. Appending two numbers is analogous to addition. For example: Append({T T} {T T T}) = {T T T T T}) is analogous to 2 + 3 = 5. The concept is renamed *addition*.
4. *Multiplication* is discovered by performing repeated addition.
5. *Divisors-of(Z)* is discovered by looking for numbers X and Y such that Multiply(X Y) = Z.
6. *Prime numbers* are discovered by specializing the concept *number* into the concept *numbers with 2 divisors*.
7. The sets of divisors of numbers was filled in. For example, the set of divisors of 12 is ({12}, {6,2}, {4,3}, {3,2,2}).
8. AM notices that each number has exactly one set of divisors that contains only prime numbers. For example, 12 contains the set of divisors {3,2,2}. This is the fundamental theorem of arithmetic: *the unique factorization theorem*.

The impressive thing about this example and about AM in general is that AM developed concepts that are analogous to numbers and then performed number theory using the same heuristics that were designed for set theory. Admittedly, set theory and number theory are closely related fields, hence one may expect that the same heuristics can be employed. Nonetheless, the unique factorization theorem is not a trivial theorem. Lenat's technique shows some promise here.

Unfortunately, AM's activity afterwards was less impressive. After 200 tasks, it was unable to find anything interesting to do. One of its last activities was to apply the operation *compose* to itself, thus creating *compose-compose.* It repeated itself and discovered *compose-compose-compose-compose:* an operation that takes 5 operations as arguments and composes them into one. It is rare that 5 composed operations would have any use or meaning. It appears that the heuristics became less relevant as discoveries were made and the environment changed.

3.5. Evaluation

Because of AM's size and complexity, it is difficult to evaluate. Therefore, many of the criticisms offered have already been broached by other critics of AM, mostly by Lenat himself [6, 7, 14, 29].

Lenat admits that AM failed to discover any new-to-mankind mathematics, but it did discover old things in new ways, and found a few interesting conjectures that Lenat did not know himself. It discovered *prime pairs* in a new way by restricting the operation *addition* to prime numbers. (For every set of 3 primes X, Y, and Z such that X+Y=Z, either X or Y must equal 2. Therefore the other two numbers are a prime pair.) It discovered *maximally divisible numbers,* a concept Lenat had not considered previously, by looking at extreme examples of numbers with divisors. Maximally divisible numbers are those that have more divisors than any of their predecessors: 1,2,4,6 and 12 are maximally divisible.

One bit of praise that Lenat gave AM was that it demonstrates an alternate way to teach mathematics--not by teaching only the finished theorems, but by a more realistic strategy of trial and error. Although this may be a laudable aspect, AM's primary goal was to discover on its own; no psychological or educational theories were adhered to in its design. Therefore, we focus on AM's ability to do actual research.

The first reaction many people have to AM is that Lenat must have cheated. They expect that the program was given too much information to start with; the discoveries were contrived to produce a specific output. Lenat claims that he gave the program only general heuristics and simple concepts, and that the discoveries prove that research can be done with a small collection of general heuristics. To back up his claim, Lenat explained how AM discovered things that he was previously ignorant of: the maximally divisible numbers and related conjectures.

Lenat makes a good point here: many unexpected results demonstrate that discovery using general heuristics is possible. This is a promising statement, but to be useful, more numerous and more important discoveries must be made. Is AM capable of that?

If one judges AM on the basis of how many new-to-mankind discoveries it made, AM fails. The only brand new discoveries it made were useless ones. For example, it created an operation that composed the operations equal and insert. The newly-composed operation takes 3 sets as input, tests the first two sets for equality and inserts the answer into the third set. For example,

equal-insert ({1,2} {3,4} {5,6}) = {false,5,6}.
equal-insert ({1,2} {1,2} {3,6}) = {true,3,6}.

None of AM's discoveries are both new *and* valuable. Below are possible reasons for this failure.

3.5.1. Domain

Mathematics has been explored for millenia by geniuses; hence, it is not surprising that a program that runs for an hour failed to find anything new. In fact, AM did remarkably well in one hour; from a basis of set theory, it discovered number theory and divisibility theory. Although it discovered only a fraction of each area, it accomplished tasks that took centuries for mankind. We examine the reasons for AM's success in more detail later.

AM was tested in another domain: planar geometry. Lenat added a few related concepts, including points and lines. Using the same heuristics, AM researched geometry, but had slightly less impressive results than it had with set theory. Geometry is also a thoroughly studied discipline. AM's success here proved again that its general heuristics have some power, but still too little to discover anything new and interesting.

Lenat wondered how AM would perform in a less-formalizable domain such as politics, but decided that a good domain is one that is easier to formalize. Furthermore, it should not have been thoroughly researched; in a new domain

AM should be able to uncover new material. Hence, Lenat tested the same principles of discovery in other domains with EURISKO; this research is described in the next chapter.

3.5.2. Concepts

What would happen if AM were given a complete set of concepts instead of the small set it now contains? For example, if AM were told about numbers and number theory, what could it discover? The consequent problem Lenat encountered was a lack of sufficient memory. But perhaps, by increasing the memory space and restricting the domain, this limitation can be overcome. If the domain is narrowed, it will be possible to supply a larger percentage of the current relevant knowledge. If, in addition, many of the related heuristics are available, the system will be at the forefront of research; the ensuing discoveries will more likely be new and useful. AM provided a valuable demonstration of rediscovery with its current concepts, but more advanced concepts and heuristics in a restricted domain might prove to work better.

Another improvement is to allow the frame structure of the concepts to be altered. While developing AM, Lenat often discarded facets that were not helping the system. Originally, AM used an intuition facet that suggested possible conjectures to be tested. Since this slot contributed nothing, Lenat removed it. If AM had this ability to remove slots, perhaps at later stages of execution it could reorganize itself so that its slots are more suitable. EURISKO has this power.

3.5.3. Correctness and Theorem Proving

A problem that Lenat mentioned repeatedly is that AM lacks theorem proving capabilities. Conjectures are made on the basis of a number of supporting examples (usually 10 or 20). If all examples satisfy a statement, the statement is offered as a conjecture and is believed by AM. Incorrect conjectures could result. For example, AM might conjecture that no numbers have more than 6 divisors after it studies the first 20 natural numbers.

In mathematics especially, it is important to have theorem proving ability, otherwise conjectures could be built upon falsehoods. Perhaps, as a solution, whenever a conjecture is made that is based on examples, AM should invoke a heuristically-guided theorem proving procedure before continuing.

One way to avoid this is for each time a conjecture is made to stop and ask the user to provide a proof or an agree/disagree response. The consequence is that the discovery program would no longer be completely automatic. This lack of theorem proving ability is not the most crucial shortcoming of AM; it seems that all of AM's conjectures are true regardless.

3.5.4. Interaction

AM's user interface is limited. There are more ways that AM could interact than by simply allowing the user to rename concepts or to prove conjectures as discussed above.

AM's path toward discovery is determined by interest values. These values are based on a shallow grasp of mathematical aesthetics, mostly concerning the abundance of the examples that can be found. It might be helpful if occasionally the user could interject and recommend a path instead. The danger of this idea arises because often while AM looks at these apparently boring areas it makes new discoveries. For example, Lenat thought that the area of numbers with many divisors would be dull and useless; he knew that prime numbers were far more interesting, but AM discovered interesting things about maximally divisible numbers that it would not have if Lenat had interfered.

Another opportunity for interaction occurs when the user thinks of a good heuristic for a certain situation, or when he thinks of a new concept that AM has not discovered but should know about. Again, if the user could give the system the extra information, the discovery process may be enhanced. EURISKO tackles some of these problems by allowing more user input.

3.5.5. Goal Direction

Related to interaction and correctness is AM's inability to search for discoveries based on a goal. If it had a goal, such as a particular theorem to be proved or a general kind of relationship to consider, the area of study might be better focussed. Much discovery is done this way. For example, mathematicians decide on theorems that they would like to prove or on specific areas of study to explore, then they build up their theory with definitions and lemmas as they strive towards the goal.

Currently, AM cannot do this. It first defines things and then looks for conjectures. A plausible goal generator module is needed, but building such a module is not trivial. Perhaps goals could be supplied by the human user.

3.5.6. Interestingness

AM is weak in its ability to judge the value or interestingness of its tasks and concepts. To make such judgements may require a broad knowledge of mathematical aesthetics and of the general usefulness of mathematical ideas in the world. Mathematicians implicitly use a deep knowledge of mathematics and the world to decide what concepts are interesting and what path of study to follow. Without this knowledge, a system may have difficulty judging interest. As with other aspects of AM, simplifying assumptions were made. For example, the value of a task is calculated by the following formula (where R_i are the reasons for the task, and A, F, and C are the Act, Facet and Concept involved).

$$Worth(Task)=\sqrt{\sum R_i^2}*(.2*Worth(A)+.3*Worth(F)+.5*Worth(C))$$

Lenat offered no justification for this formula except that it works surprisingly well.

Another simplifying assumption is that task-rating reasons are independent of each other. For example, the task *look for examples of sets* might have 2 reasons for its importance:

examples of Sets would permit finding examples of Union
examples of Sets would permit finding examples of Intersection

The agenda treats these reasons as mere tokens with values. It does not understand their meanings, hence it does not notice that the two meanings are closely related; thus the task gets a higher value than it should have.

Lenat performed an experiment with AM where he started the program with all concepts having the same value. The experiment was to test how important the initial values were in determining the search path. To his surprise, AM made most of the same discoveries. After a slow start, the concepts eventually attained similar values, thanks to the heuristics that notice patterns to increase the value of concepts.

Another experiment was to remove the agenda mechanism so that instead of choosing the highest-rated task, tasks are chosen for execution at random. The consequences proved to be severe; AM discovered little of interest. The conclusion is that some power exists in Lenat's approach.

3.5.7. Representation and Automatic Programming

When Lenat studied AM in retrospect to find out why it worked as well as it did, he made an interesting observation: its basic activity was to search through a space of LISP functions. The definition of each concept is a LISP function, and the heuristics that create new concepts do this by mutating the function. For example, the concept ***list-equal*** is defined in the following way:

```
(lambda (X Y)
  (cond ((or (atom X) (atom Y)) (eq X Y))
      (t (and
           (list-equal (car X) (car Y))
           (list-equal (cdr X) (cdr Y))))))
```

Its meaning is as follows:

```
X equals Y if
    ((X and Y are both atoms and are equal)
       OR given neither are atoms,
    (the first element of X and Y are equal
    AND the remaining lists are equal))
```

At some point, AM wanted to generalize this concept. There are three ways it could have done this: it could drop either one of the two conditions around the AND, or it could change the AND to OR. One of the newly created concepts is the following:

```
X equals Y if
    (X and Y are both atoms and are equal
       OR given neither are atoms,
    the remaining lists are equal)
```

This definition strips off the first atoms of X and Y one at a time. If the sets become empty simultaneously, they are deemed equal. Therefore, this is simply a definition for the predicate *same size;* it returns true if X and Y are of identical length.

By mutating a concept's definition, AM is automatically programming new concepts. The conclusion that Lenat reached is that AM's success with a simple notion of automatic programming depended heavily on the representation of the concepts. Because LISP was designed to have a close affinity with mathematics, relation, small mutations of LISP code often lead to interesting mathematical concepts. Lenat saw this as AM's strength.

The consequence for other domains is that more thought must go into the representation if similar code manipulation is carried out. He addressed this problem when he attempted discovery in other domains with EURISKO.

3.5.8. Heuristics

Ultimately, the heuristics are AM's power and greatest limitation. Without them, AM could do nothing; but the limitation lies in the inability to find the best possible ones, or at least those necessary to discover.

A heuristic that says *look at the extreme examples of a concept* seems to be a good heuristic because it is often similarly employed and is crucial to many of AM's discoveries. Yet we cannot say whether it will eventually be used to make new discoveries or whether it was similarly employed by mathematicians.

Perhaps this heuristic is ahead of its time; that is, perhaps most of the concepts AM discovered using it were originally discovered by other means before anyone thought of the heuristic itself. This is the criticism of *cheating* again. Perhaps even though Lenat did not plan any particular discovery paths and did not foresee how each heuristic would come into play or if they would be used at all, he still cheated by choosing heuristics ahead of their time. The heuristics might in a subtle way embody the breakthroughs needed to make certain discoveries.

A related criticism is that AM relies heavily on a human to impart meaning to its work. For example, when AM canonized *sets* to form *sets of T's,* Lenat had to relate concept back to the world. He realized that such a set is analogous to numbers and thus gave it that name. Such interpretation is a necessary part of automatic concept formation, so the problem cannot be eradicated completely, but the danger is that some discoveries could be misinterpreted or overlooked entirely by the user.

How can these heuristic limitations be mitigated? To avoid the criticism of cheating, we might omit the offending heuristics, but that would only hurt the program's ability to discover. We need more heuristics, not fewer. The production rules we acquire must be complex enough to deal with the more complex ones that AM synthesized.

How does one find new heuristics? Alas, this is a discovery problem itself. Heuristics are needed to help us find heuristics. This problem seems much harder than the original problem of discovering mathematics, but Lenat tackled it too with EURISKO.

3.6. Conclusions

AM is a valiant first attempt at discovery learning. Much was learned about the possibility of automating the discovery process. AM makes the idea seem possible by suggesting a general method, but it also points out many difficulties.

4. EURISKO

EURISKO [6, 7, 9, 16, 27-29, 38] was conceived by Lenat in 1976 and is still under development. It is an attempt to address some of AM's weaknesses; the most crucial being the inability to discover new heuristics. As EURISKO discovers concepts it also monitors its progress and develops new heuristics by using a meta-level discovery system.

Progress with EURISKO was not easily attained. Lenat tried to incorporate AM-like techniques in EURISKO's meta-level. He tried to manipulate the code of heuristics with other heuristics. When this failed, he surmised that the space of heuristics must be fundamentally different than the mathematics space, and a wholly new approach is necessary.

Later, he realized that the failure was not due to any peculiarities in the space of heuristics, but was due to representation. He noticed that LISP was a good language for simple mathematics concepts. As explained in the previous chapter, a syntactic change was related closely to a semantic change because of the ties engineered between LISP and mathematics. Each heuristic, however, consisted of several lines of LISP code, split into if-parts and then-parts. A syntactic change usually meant a LISP error or a useless heuristic. The intended meaning of the mutation was rarely accomplished.

The solution to this problem was to create a new language that is a natural representation of heuristics, where a code mutation would result in the intended change of meaning. He accomplished this with the LISP-based language RLL (Representation Language Language) [16].

RLL expedites the definition of frame slots that correspond to descriptive properties that heuristics might possess. The code stored in a slot is high level and is translated into a larger LISP program. The small size of the piece of code in RLL allows it to be manipulated meaningfully.

Figure 4.1 shows two representations of the same heuristic, first in LISP, then in RLL. The heuristic's meaning is *If you want to find examples of a set B, and you know some function f: A -> C, where C is known to intersect some generalization of B, then apply f to examples of A and collect the results.*

```
LISP:  IF:  (AND (EQ Cur-Action 'Find)
                 (EQ Cur-Slot 'Examples)
                 (MEMBER 'Collection (Isa Cur-Concept))
                 (SETQ f (SOME (Examples 'Function)
                        '(LAMBDA (g)
                        (Doesintersect (Range g)
                               (Generalizations Cur-Concept))))))
        THEN:  (SUBSET (MAPCAR (Examples (Domain f))
                   '(LAMBDA (x)
                        (APPLY (Alg f) x))
                   '(LAMBDA (e)
                        (APPLY (Defn Cur-Concept) e))))

RLL:    IfCurAction:        Find
        IfCurSlot:          Examples
        IfCurConcept:       a Collection
        IfForSome:          a Function f
        IfIntersects:       (Generalizations CurConcept) (Range f)
        ThenMapAlong:       (Examples(Domain f))
        ThenApplyToEach:    (Alg f)
        CollectingIfTrue:   (Defn Cur-Concept)
```

Figure 4.1: two representations of a heuristic

Lenat claims that RLL has a meaningful and deep relationship with heuristics [19]. RLL's practical power is manifested in EURISKO's ability to derive new heuristics.

Several other changes were made to AM in producing EURISKO. To allow the representation to be even more suitable for heuristics, Lenat made RLL flexible. Ideally, a system should be able to change its representation language when it notices that the language is unsuitable. EURISKO takes a step towards this goal by allowing manipulation of the frame slots. Slots can be both created and removed. To make this possible, each slot has an associated frame that supplies information about it. This slot-frame can store information about the slot's history, and can change the slot when it sees fit.

The agenda is treated the same way. It is no longer just a simple queue, but can be split apart and rejoined as necessary. A special purpose agenda-frame supplies information about the agenda's status. Usually the agenda would be split up into groups of tasks that relate to a particular topic so that EURISKO can focus on a single topic of research. An example of such a heuristic is the following: *If agenda A contains more than four times as many tasks as the average agenda, then (try to) split A into about three pieces.*

The other major change in EURISKO is the enhancement of interaction with the user. EURISKO allows the user to interject by adding concepts or tasks and by choosing paths of discovery to follow. It also keeps a limited model of the user so that it can respond accordingly. For example, if EURISKO knows that the user is a mathematician, it responds to suggestions from the user almost immediately. When an AI researcher is using the system however, it puts the request on the agenda and may wait a long time before performing it. Lenat explained that mathematicians would rather contain full control whereas AI researchers enjoy a program that is in command [14].

A final major difference between AM and EURISKO is that EURISKO is able to do research in several domains. Not only did it work on mathematics, it also was put to use in VLSI design, game playing, LISP programming, knowledge representation, cleaning oil spills, evolution simulation and heuretics. We discuss some of the results in the next section.

4.1. Results

4.1.1. Naval Fleet Design Game

Traveller Trillion Credit Squadron is a game that involves designing a set of navy fleets based on a large list of constraining rules. The fleets are pitted in battle against each other to determine the best design.

A tournament was held in California. After 6 rounds of single elimination, EURISKO emerged as the winner. The following year, 1982, the game rules were changed somewhat; EURISKO designed a new fleet that again proved to be the best.

To design a fleet, each player is given a trillion credits (dollars) to spend on parts of ships. The design is a balance of tradeoffs. First a player must design individual ships by specifying the weapons, armor, agility, amount of fuel, and other features. The ships are then united into a fleet that satisfies further constraints concerning the number of ships, total weight, and ability to act as a unified force.

Lenat programmed many new frames especially for this game. Most of them kept information about specific items of the design. For example, figure 4.2 is the energy gun frame.

One of the new frames represented is the activity *play a game*. Playing a game involves three sub-activities: designing a fleet, simulating a game, and abstracting new design heuristics. Lenat crafted heuristics especially to deal with each of these sub-activities.

These heuristics are in turn monitored by meta-heuristics that may from time to time change the heuristics used at the lower level. The meta-heuristics also monitor themselves and have potential to change their own code.

The fleet that won the tournament in 1981 was peculiar in many ways. It had many extreme features: small ships, many of them, agile ships with no armor, and lots of small weapons. One of the heuristics that EURISKO synthesized was this: *In almost all Traveller Trillion Credit Squadron design situations, the right decision is to go for a nearly, but not quite, extreme solution.* EURISKO won the tournament by finding loopholes that such extreme designs exploited.

Name: EnergyGun
Generalizations: (Anything Weapon)
AllIsA: (GameConcept GameObj Anything Category
    DefensiveWeaponType OffensiveWeaponType Obj
    AbstractObj PhysGameObj PhysObj)
IsA: (DefensiveWeaponType OffensiveWeaponType)
MyWorth: 400
MyInitialWorth: 500
DamageInfo: (SmallWeaponDamage)
AttackInfo: (EnergyGunAttackInfo)
NumPresent: NEnergyGuns
UspPresent: EnergyGunUSP
DefendsAs: (BeamDefense)
Rarity: (0.11 1 9)
FocusTask: (FocusOnEnergyGun)
MyIsA: (EURISKOUnit)
MyCreator: DLenat
MyTimeOfCreation: 4-JUN-81 16:19:46
MyModeOfCreation: (EDIT NucMissile)

Figure 4.2: a EURISKO frame

One taint on EURISKO's performance is that it had human help to design the winning fleet. Lenat estimated that his interaction with EURISKO accounted for 60% of the victory. He had monitored EURISKO's performance and changed heuristics and concepts all along. At one point, EURISKO designed a small agile ship that was almost unhittable by the opponent. Such a ship could be used as a stall tactic while the rest of the fleet was being repaired. EURISKO overlooked the importance of the ship; Lenat had to raise its worth value himself.

He still claims that he could not have won the tournament by himself. EURISKO's contribution was necessary. He also claims that other exhaustive methods or monte carlo methods would fail in this situation; the search space is too large. EURISKO's ability to search the design space with some rudimentary knowledge that compares fleets and extracts principles to explain why one fleet is superior, contains power to find a good design.

4.1.2. 3-Dimensional VLSI Design

Because of a new development in VLSI fabrication techniques, Lenat was given another area in which to test EURISKO. Gibbons developed a way to build high-rise chips, VLSI chips with multilayers of silicon [26]. Since, there were no 3-D VLSI design experts in the world yet, EURISKO had the opportunity to become the first. Whatever EURISKO discovered would be new.

The basic method of discovery was much like that in the domain of navy design. One of the concepts was an activity that designed a microcircuit and evaluated it. Heuristics performed each aspect of design and evaluation, while meta-heuristics created better heuristics based on the utility of the current heuristics.

Before the program could proceed, Lenat made several design decisions concerning the domain. He needed a formal way to represent primitive microcircuit elements, operations to combine primitive elements, and criteria for evaluating new structures. After much study of the problem he decided to represent each type of material (n-doped, p-doped, metal) as a three dimensional rectangular tile and to represent combinations of elements by stacking tiles on top of each other.

To combine elements, Lenat used a variety of heuristics including both implausible path pruners and plausible space generators. Some of the heuristics were very specific; Lenat hoped to make genuinely new discoveries so he did not intentionally withhold any heuristics. He included some strategies from other domains such as examining extreme cases and looking for patterns.

To evaluate the planned structures, EURISKO had heuristics that interpret a pile of tiles to find the outputs of the device it models. The structure is evaluated on the basis of its outputs and how it fits into a larger system. Again meta-heuristics monitored the whole process. For example, when several successful designs resulted from a heuristic that stacked tiles symmetrically, symmetry was rated higher and was used in more design situations.

The results were apparently successful. The symmetry heuristic contributed several useful designs, including a logic gate of seven tiles that has both a NAND and an OR output. On the other hand, many anomalies were discovered; one was a flip-flop built on a mobius strip. It worked but was too expensive to build.

Lenat is continuing with research in this area. He is planning to incorporate a graphic display screen so EURISKO's designs can be visualized by the user as EURISKO tests them. He is also experimenting with different shapes of tiles so fewer limitations on the designs will exist. He expects that important designs and design-heuristics will be discovered. His optimism is obvious in the following quotation:

> Our final remark is a strategic one on the possibility of major industrial or national impact if this AI application can be successfully pursued. As we enter the era of VLSI technology, there are shortages of critical people, an explosion of design complexity, and increasingly aggravating test requirements, to name only a few of the problems hindering the field. The advent of three-dimensional VLSI technology explodes the magnitude of all of those problems. Any industrial firm or nation which could successfully devote a large number of computation cycles on a sustained basis to intelligent exploration of microelectronic design, fabrication, and test possibilities would certainly be ahead. We enjoy the dream. [27]

4.1.3. Mathematics

With EURISKO's apparent success in these two domains it is interesting to consider how it performed in AM's domain of elementary mathematics. EURISKO was given the same starting concepts as AM, and a set of 50 heuristics that subsumed AM's original 243 (Lenat observed that many of AM's specialized heuristics had no more power than their generalizations; these were omitted.) The results were disappointing however. Although EURISKO rediscovered the same things AM did in slightly less time, it did little else. Lenat suggested that mathematics is too thoroughly researched for EURISKO to have success. Some new heuristics were discovered, other already known concepts were rediscovered, and new slots were added to heuristics, but the overall results were disappointing. Another reason for failure is that EURISKO's knowledge is organized and represented too rigidly; the representation is still too inflexible.

4.2. Evaluation

EURISKO is more complex than AM; it is larger, runs longer, has broader goals, and is harder to understand. To critique it, we first discuss heuristics in general to see how they derive their power. Again most of the remarks here are based on Lenat's own criticisms.

In his papers on the nature of heuristics [27, 28, 29], Lenat studied the power of heuristics and the heuristic theory that EURISKO was modelling. Basically, heuristics are compiled hindsight. That is, they are rules of thumb that arise by performing an action in some situation and monitoring the results. If the action is appropriate for the situation, the same action or a similar action will likely work in similar situations. Implicitly, when we use rules of thumb, we are assuming that there exists some consistency in the world with respect to the appropriateness of actions in given situations. Of course, the world is not constant everywhere, so heuristics often fail, but it is consistent enough so that to use heuristics is itself a good rule of thumb.

EURISKO assumes that its domains contain regularity and that additional appropriate heuristics can be created by specializing, generalizing, and making analogies of known heuristics. Furthermore, it assumes that as work progresses, and gets further and further from the original situation, the heuristics might have to be very different from the original heuristics. The meta-heuristics used to build new heuristics might also have to change as the heuristics change to work in the new situation; therefore meta-meta-heuristics are needed. Lenat assumed that meta-heuristics and meta-meta-heuristics are similar enough so they can successfully be implemented by using only meta-heuristics that can manipulate their own code. Thus, an infinite hierarchy of meta-levels of heuristics is avoided.

Another element of this study is that as heuristics and situations change, new representations are needed in order for syntactic manipulation to work. EURISKO accommodates this idea by allowing new slots to be defined in the frames. It does not allow a complete change of representation however.

Lenat summarized his theory of heuristics in the following 7 statements:

1. A set of heuristics can guide concept discovery.
2. A new field will develop slowly if no specific new heuristics for it are concomitantly developed.

3. Heuristics can be used as plausible move generators or as implausible move eliminators.
4. The generalization/specialization hierarchy of concepts induces a similar structure upon the set of heuristics.
5. Heuristics are compiled hindsight.
6. The space of 'domains of knowledge' is granular (not completely continuous).
7. Use heuristics to decide which heuristic to apply next.

EURISKO models all aspects of the theory to some degree. In fact, the theory was developed by studying how EURISKO works, but it seems to embody a realistic view of the world. Its underlying assumption is that heuristics derive their power from regularities in the world; EURISKO exploits these.

4.2.1. Representation

Is EURISKO using the best representation for its heuristics? As discussed earlier, the representation ought to suit the domain. EURISKO allows frames to differ among domains. Although this seems to be an adequate approach, there may be a more flexible, but still feasible, solution.

Related to this is the problem of meta-heuristics. Lenat decided that all meta-heuristics ought to be represented in the same way; nay, that all meta-heuristics are identical in all meta-levels. Would it be better to allow for alternate representations for different meta-levels or at least to distinguish between heuristics on different levels? Admittedly, to allow such distinction on even a few levels would create an infeasible problem. Somewhere a line must be drawn; Lenat conveniently drew it at one level.

Another problem is that in each domain, a unique representation might be most convenient. A human must abstract features from the real domain and find a representation for them; EURISKO cannot do this itself. This deficiency is understandable, but it should be noted that human work is still required.

4.2.2. Discoveries

EURISKO's actual discoveries are partial proof that it is a useful system. Many of the discoveries are tainted however. In the Navy design domain, Lenat had to inform EURISKO of its discovery. EURISKO would generate an interesting design and misunderstand its value. Lenat had to step in to insure that the discovery was not discarded. This is the opposite problem that was mentioned in the previous chapter. It was suggested there that the user might not be able to appreciate a discovery because of his lack of conceptual perspective. EURISKO demonstrated that the converse is true: the program lacked the conceptual perspective with which to understand its own discoveries.

The lack of success in the mathematics domain is disappointing. When Lenat isolated the problems of AM and then tackled them in EURISKO, it was hoped that EURISKO could research mathematics further. But, although EURISKO was successful in other domains, it failed to make progress here. Is this problem a problem of representation? Can we find a representation where research can be done on more complex mathematics, or is EURISKO inherently only capable of simpler domains?

4.2.3. Miscellaneous Comments

Some ideas that EURISKO uses seem good, but the implementations may not be the best. For example, EURISKO allows for interaction with the user. Does such interaction facilitate discovery? We consider domains where new discoveries are being made since when asking this question in rediscovery domains, such as mathematics, the user knows too much; his interaction could give the program an unfair advantage. Interaction with the navy design game was very important, and by Lenat's own admission, was mostly responsible for the success. It is a good idea to allow such interaction; currently it appears unrealistic to expect the whole problem to be solved by computer.

EURISKO employs a hill-climbing process and, therefore, is susceptible to all the associated problems such as stranding itself on ridges or plateaus. It is unclear how well EURISKO handles these problems. In designing fleets, for example, what if the manipulations that EURISKO can make on its current fleet are too small to get off of a ridge and eventually find the best possible ship? Perhaps somewhere in the space of fleet designs, there is a class of designs that can defeat EURISKO's design but EURISKO was unable to find it because it got stranded on a local fleet-design optimum.

4.3. Conclusions

Most of these criticisms are very difficult problems that we cannot expect to solve simply. EURISKO should be praised for leading to a better understanding of such problems that are inherent to many areas of AI. The difficulty in criticizing EURISKO testifies that the work is remarkably good. Even though its true discoveries are few, it has clarified the inherent problems.

EURISKO's heuristics derive much of their strength from a successful combination of ideas. Its heuristics create specializations, generalizations, and analogies; some generate a plausible space and others prune the space; all these in combination are integral to the success. More research needs to be done in the field of heuretics, but the results of EURISKO have brought us a long way.

5. BACON.5

BACON.5 (hereafter referred to as BACON) is the fifth in a series of discovery systems developed by Langley [4]. It takes an alternate approach to the discovery process than AM and EURISKO. Furthermore, it is applicable in a different domain; it discovers empirical laws. Examples of the physics laws that it rediscovered include Snell's law of refraction, the conservation of momentum, Black's specific heat law, Joule's formulation of energy conservation, the ideal gas law, Kepler's law, and Ohm's law.

BACON is easier to understand than AM and EURISKO because it has a clearer goal (to discover constancies in data), and obvious limitations (for example, it cannot carry out experiments in a formal system as AM does). To find empirical laws, BACON is given a set of data from various physical experiments. BACON does not contain a model of the world in which it attempts to discover; rather, it is supplied with the values of variables that are used to specify the law. To discover the ideal gas law, BACON was given data from several experiments with the measurements of pressure, volume, temperature and number of moles of gas.

BACON employs 70 heuristics that guide its search. The heuristics find patterns in the data including constancies, linear relations, and monotonic trends. Other heuristics postulate intrinsic properties or reason by analogy. BACON combines variables into products and quotients until an invariant term is found. The following example, the discovery of Kepler's Law, illustrates how BACON works. Kepler's law states that the period of a planet's revolution around the sun, p, is related to its mean distance from the sun, d, by the equation $d^3/p^2=k$ for some constant k. (Note that the numbers in the example are simplified for ease of understanding.)

Features		
Planet	*p*	*d*
Mercury	1	1
Venus	8	4
Earth	27	9

The above table constitutes the input to BACON. First BACON notices that both variables are monotonically increasing, but are not linearly related. Therefore a heuristic rule fires which creates the term d/p in an attempt to find a constancy. Now the data is as follows:

Features			
Planet	*p*	*d*	*d/p*
Mercury	1	1	1.0
Venus	8	4	.5
Earth	27	9	.33

Eventually BACON notices that the entries for d and d/p are monotonically increasing in opposite directions. To attempt to find a constancy, it multiplies the terms to form d^2/p. The data is now as follows:

Features				
Planet	*p*	*d*	*d/p*	d^2/p
Mercury	1	1	1.0	1.0
Venus	8	4	.5	2.0
Earth	27	9	.33	3.0

The same heuristic is fired again when it notices the opposite monotonic increase of d/p and d^2/p. The final table is as follows:

Features					
Planet	*p*	*d*	*d/p*	d^2/p	d^3/p^2
Mercury	1	1	1.0	1.0	1.0
Venus	8	4	.5	2.0	1.0
Earth	27	9	.33	3.0	1.0

Now the constancy heuristic fires because the invariant d^3/p^2 is noticed. Thus the law is found. The actual execution of BACON is more complex than this as there are many more possible paths it can pursue in its search.

5.1. Evaluation

BACON seems to be more clearly defined than AM and EURISKO because it is possible to state concretely what its limitations are. Unfortunately, these limitations are severe. Langley described many technical difficulties with BACON; we discuss these in the following sections.

5.1.1. Experiments

BACON cannot suggest experiments. The user must be able to guess which experiments are necessary for BACON to find laws. Once the user decides on the type of experiment, BACON may enforce constraints. For example, BACON needs many experiments where only one of the relevant variables is varied. To discover the ideal gas law BACON compared values of experiments that had constant values for temperature and number of moles. Once it found the relationship between volume and pressure, it looked at experiments where the temperature varied but the number of moles stayed constant. A proper set of data is essential.

A second problem is that BACON is sensitive to the order of the experiments. It makes partial hypotheses already after looking at the first bits of data. Then it considers the other data with the partial hypotheses in mind. If the first data is misleading, BACON could pursue an inappropriate path in its search.

Third, BACON handles noise poorly. Previous versions of BACON were even less successful in this regard. The current version employs numerical methods that find approximations, but noise is still a problem. One enhancement that Langley hopes to add to BACON to alleviate this problem is the ability to keep multiple hypotheses and then suggest experiments that would help decide between them.

Fourth, the user must decide which variables are pertinent to the ensuing discovery. This implies some cheating; one has to know something about the discovery before the discovery is made. Alas, BACON cannot handle irrelevant variables.

5.1.2. Domain

BACON is limited by its domain of applicability. It is best at finding relationships between numeric variables. It has some powers to find patterns in letter sequences such as TETFTG..., but not ones as difficult as ABTCDSEFR... It has limited power in dealing with non-numeric data with intrinsic properties. For example, it discovered Snell's law of refraction, including the intrinsic property *index of refraction* for water, oil, and glass.

Furthermore, the form of the constancies it can find concerning numerical data is limited. BACON can find product and power laws, linear relations, and complex combinations of these forms. However, it does not understand trigonometry. Langley hopes to attain more complex functions in future versions of BACON.

5.1.3. Discovering

It is unclear whether BACON is able to discover anything new. Langley makes no mention of how BACON will eventually be used. It seems that there are two alternatives: it could be incorporated within an AM-like discovery system, or it could stand by itself.

The biggest difference between AM and BACON is that AM discovers in a formal system whereas BACON discovers in empirical domains. That does not imply, however, that BACON cannot be used within a system such as AM. In fact, AM already uses some of BACON's techniques to find relationships between various mathematical concepts.

How BACON could be used by itself is harder to imagine. Physicists and other scientists could use it, but it would be a rare occurrence, and it would probably be of little help. Because BACON does not help the scientist to design experiments or to choose among variables to measure and consider, even before BACON comes into play, the scientist will likely have already made the important breakthroughs. Langley explains:

> [BACON] addresses itself to that part of the scientific process that starts with a set of data and seeks to discover regularities in them. Of course, this is only one of several ways in which scientific discovery takes place, and only one of the steps (albeit a crucial one) in data-driven discovery. Scientists must also determine what data to gather,

> must invent and use appropriate instruments for making observations, must draw out potentially observable consequences from theories, and must analogize from one phenomenon to another. Thus BACON's capabilities provide an account of one important part of the scientific enterprise, while leaving other equally important components to future investigation.

Hanson's account of Kepler's discoveries concerning planetary orbits [18] render BACON even less relevant. For Kepler, the most important breakthrough was the realization that planets travel in elliptical patterns, since it was previously assumed that they travelled in circles. His other discoveries, including the one described earlier, were trivial in comparison. Thus, although Kepler used data in his research, BACON would most likely have been unhelpful to him.

6. Aref's System

The final system we critique is a smaller, newer, and nameless system by Aref [1, 2]. The system is based on AM's modified production system architecture, but has a different domain: data structures applications.

```
Name: lessp
Isa: one-to-one
Definitions:
        Semantic: return true if NUM1 < NUM2
        Parameter: NUM1, NUM2
        Code:
                Precondition: NUM1 and NUM2 are numbers
                Relation: NUM1 is less than NUM2
Examples: (2 5), (22 202), (73 83)
Worth: 100
```

Figure 6.1: a relation frame: less than

There are many similarities between Aref's system and AM. Aref's has an agenda mechanism to order the tasks based on interestingness, it uses heuristics as plausible space generators, and like AM, its knowledge base is stored in LISP frames.

However, the differences between the two systems are vast. The most obvious difference is the knowledge base. It is divided into three separate groups, each with a different frame structure to represent concepts. The largest of the three groups is the relations. Seven of them are supplied at the start. These are equivalent to the LISP predicates *oddp, numberp, memq, less, alphaless, atom,* and *listp* [39].

The other two groups start with one frame each. The lone structure is *set;* a structure containing LISP atoms (numbers or character strings).

Name: set
Isa: structure
Definitions:
  Semantic: a collection of non-repeated elements
  Parameter: L
  Code:
    Precondition: L is not an atom
    Termination: the first element is not an atom; and there are no more elements in L
    First-step: the first element is an atom; and the first element is not in the rest of L
    Recursive: the rest of L is a list
Examples: (kxfup c s m 189 219 788 771), (777)
Worth: 200

Figure 6.2: the structure frame: set

The only operation is *search;* it performs a sequential search through a set to find an element.

The heuristics are divided into the same three groups. There are 26 heuristics altogether. Of these, 15 are used to add tasks to the agenda or to reorder tasks. The remaining 11 perform tasks and create new concepts. Three of these create new relations, five create new structures, and three create new operations. These are described below; each with an example to clarify.

Name: search
Isa: operation
Definitions:
  Semantic: looks for an element in a structure
  Parameters: OBJ, LST
  Code:
    Precondition: OBJ is an atom; LST is not an atom
    Succeeds-if: the first element of LST is an atom; and the first element of LST equals OBJ
    Fails-if: there are no elements in LST
    Recursive-part: search for OBJ in first element of LST; search for OBJ in rest of LST
Domain: Structures
Worth: 200

Figure 6.3: the operation frame: search

6.0.1. Relation Heuristics

Invert relation: adds the operator 'not' to a relation definition. 'Less' becomes 'greater than or equal to'.

Generalize relation: changes the domain of the relation from 'one-to-one' to 'one-to-many'. 'Less(3 6)' becomes 'Less-than-all(3 (5 8 7 9))'.

Link relations: composes two relations such as 'less(16 23)' and 'greater(38 23)' to form 'less-greater(23 (16 38))'.

6.0.2. Structure Heuristics

Add a relation: inserts a relation between elements of a structure. Adding 'less' to 'list of numbers' forms 'ordered list of numbers'

Change a relation: replaces the relation in a structure. Changing 'numberp' to 'oddp' in 'list of numbers' gives 'list of odd numbers'.

Nesting: each element of the structure recursively becomes a structure of the same type. 'Pair' becomes 'binary tree'.

Generalize: removes a condition in the definition of a structure. 'Set' may have duplicate elements and becomes 'list'.

Specialize: adds a condition to a structure. 'List of numbers' becomes 'list of 3 numbers'.

6.0.3. Operation Heuristics

Change domain: the search is performed on a new domain. 'Search list' becomes 'search ordered list'.

Domain Division: the list is split and the search is done on the two halves separately. 'Search' becomes 'search by division'.

Domain Testing: a test step is added to the search algorithm. A search on 'ordered list' is now performed only if the atom being searched for is not less than the first element or greater than the last element of the list.

Unlike AM, instead of attaching heuristics to the slots of the concepts to which they are relevant, the heuristics are stored in a separate tree structure. Each task specification contains a list of two or three words which completely specifies the heuristic needed. For example, one task was [259 List-of-numbers Add-relations One-to-one], which says *add a relation between elements of the concept list-of-numbers.* The proper heuristic relation*adda* is easily found using this specification.

The difference between AM's and Aref's method of judging interest is great. Aref uses a simpler method; the worth value of a new concept is a fraction of the worth of the concept used to create it. The fraction depends on which heuristic was used. Tasks are rated in a similar way. There are a few complications: if a task fails it is returned to the agenda with a lower value, and if a task is suggested twice the two values are summed. No attempt is made to give reasons for suggesting tasks or to evaluate concepts on their own merit as AM does.

6.1. Results

The most interesting rediscoveries of the system are ***binary search, ordered binary trees,*** and ***search ordered binary trees.*** To understand how these discoveries were made, the discovery process is explained in three parts: discovery of relations, discovery of structures, and discovery of operations.

The system tended to execute the following loop:

> modify search to work on new structures
> modify the structures

build new relations to be used in modifying structures

The system did not always follow this loop because of the intricacies of the agenda mechanism, but all new operations were obtained after a sequence of modifying the relations and the structures.

6.1.1. The Relation Space

Since the heuristics and frames were divided into three groups, each of the groups expanded on a separate search space. In the relations space the discoveries were few and simple. Being limited to seven starting relations and three heuristics the system had a small space in which to work. The space was further constrained because each heuristic could only be applied once to any relation, and because some relations and heuristics were incompatible. *Less* and *alphaless* were the only relations that *generalize* and *link* could be applied to, and apparently only one of them could be applied at a time. Some of the discoveries made are these: *even* (by inverting odd), *greater* (by inverting less), and *all-greater* (by generalizing greater).

6.1.2. The Structure Space

The structure space was more complex because all of the relations found in the relations space could be used to create new structures. The original structure was a set of atoms. *Change relation* replaced *atom* with other unary relations, *number, odd,* and *even,* to create the concepts *set of numbers, set of odd numbers,* and *set of even numbers. Add a relation* could add any of the binary relations to the above structures. By adding *greater than* to *list of numbers, descending ordered list* was discovered. The power of the heuristic was more evident when a tertiary relation was added. *Ordered forest* was found by adding the relation *less-greater-pair* to *list-of-numbers.* For example, ((23 (22 234)) (809 (100 958)) (2 (3 45))) is an ordered forest.

Specialize turned out to be a useless heuristic. It could add the number 'two' or 'three' to the specification of a structure, thus creating structures such as *set of two numbers* and *set of three odd numbers. Generalization* was also a weak heuristic; it discovered *set* from *list* by removing the condition that no element appear twice.

Nesting was more powerful. It gave a recursive definition to a structure, thus creating *binary search tree* from the above *ordered forest*.

6.1.3. The Operations Space

This space was simpler than the structures space. Although *search* could be applied to any of the structures developed in the structures space, most of the structures were of the same form. The two common forms were lists and trees. Elements within those forms were numbers or atoms, and were ordered or unordered. To find new search algorithms, all that could be done was to take on a new structure, to divide the structure and search each part separately, and to add a testing step to aid the search.

Adding a testing step was successful only if the structure contained ordered elements, and dividing the domain was successful only if the structure was a list (as opposed to a tree). The discovered operations *search a binary search tree by testing* and *binary search a list* seem to be the only interesting operations in the space. There are many variations of these depending on the kinds of elements in the structure, but the algorithms themselves are the same.

6.2. Evaluation

From the small size of the search space we can see that this system has serious limitations. In summary, there are about 20 simple relations possible; the structures are restricted to lists and trees with various constraints on the types of elements, the size of the structure, and the ordering of the elements; the possible operations are search applied to lists and trees, with the possibility of using a binary search technique. Although not exhaustively explored, it is unlikely that the space defined by the current heuristics contains any interesting discoveries.

The questions we should ask are many: what accounts for this inability to discover? What can be added to make it work better? In what way is the basic design hurting or helping the discovery process? Many of the answers to these questions are discussed by Aref [1].

6.2.1. Heuristics

One problem with the heuristics is that there are too few of them. Since heuristics are plausible space generators, more of them, or new ways to apply them, are needed to encompass a large space containing more and better discoveries. As Aref suggested, heuristics could be added so that sorting algorithms or data storage and retrieval can be researched. The lack of meta-heuristics was not a limiting problem here; more initial knowledge and heuristics are needed first before meta-heuristics would be useful.

We evaluate the heuristics in two ways: problems with the current heuristics and heuristics that should be added.

6.2.1.1. Current Heuristics

The major problem with the current heuristics is that they are too specialized. Some are specially designed to be applicable to the system, but will not be useful if the system is expanded. The two heuristics used to form new operations *divide domain* and *add test step,* are both overly specialized. *Divide domain* seems to be a useful heuristic; it is a specialization of *divide and conquer* which has proven to be effective in algorithm design. Aref's version, however, can merely divide a list into two or three pieces to allow searching to be done on each piece separately. Its limitation is reached when applied to a tree structure. A different algorithm is needed. By incorporating better automatic programming techniques, a more general divide and conquer heuristic can be implemented.

Aref admits that the heuristic that adds a testing step is too specialized. It is able to add a testing condition to search algorithms for lists and trees, but it is unlikely that the same code manipulation will work on slightly different concepts. The general idea of testing before a search is good, but additional heuristics are needed to allow for a more general method of adding a test step.

Two other specialized heuristics are *link relations* and *nesting. Link relations* forms the *less-greater-pair* that is used by *nesting* to form *binary-search tree.* The two heuristics have power, but judging by the number of times they are used in a typical run of the system (twice each), they are too specialized.

Another problem with the current heuristics is that some of them are weak. The weakness of *invert-relation* is manifested by the shallow concepts that it creates.

Specialize structure has a similar problem. Constraining the size of a structure invariably gives rise to a shallow concept.

6.2.1.2. Heuristics Needed

To automate discovery, it seems wise to incorporate current knowledge and methods that are used by humans to do research. A heuristic that specializes trees into binary trees is likely to have success because a binary tree is an extreme kind of multiway tree. As AM demonstrated, extreme characteristics often lead to useful concepts. They led to the discovery of prime numbers, since primes have an extreme number of divisors. However, researchers are able to quickly consider more than just the extreme cases. For example, they can study a multi-way tree. By using algebraic techniques, they can decide on the optimal number of branches for a given application. Such capabilities are essential to research, but require knowledge that is lacking in the system.

Related to this is the ability to study the complexity of algorithms. Currently, search algorithms are tested on data and an average running time is produced to give an indication of the complexity of an algorithm. The method is weak since it is data dependent. Techniques for studying algorithmic complexity exist; some of these ought to be employed here.

Aref raises the question of how sorting algorithms can be discovered. Some sorting algorithms such as *bubble sort* and *selection sort* seem almost self-evident, but *heapsort* and *quicksort* do not. Heuristics that can discover such concepts should be found to enhance the system.

Even a simple algorithm such as binary search does not reveal the heuristics used to discover it. It is common sense to use binary search to find a name in a telephone book; therefore it ought not take an expert to find the same algorithm for computer science. Yet, the heuristic is hard to discern. It seems unlikely that the three steps *sort the data, search by division,* and then *add a test step* were used when binary search was first encoded. Since all three steps are crucial to the conceptual idea, perhaps all three were incorporated into one heuristic. Such a combined heuristic seems too specialized for our purposes though.

The overall method of research by humans appears different; when looking for an algorithm, they first have a goal in mind of what they want it to do, then they design the algorithm, and finally decide on the data structures that are necessary to support the algorithm. Aref takes the opposite approach. Perhaps some of AM's simple heuristics are applicable, but generally they are more suitable for concept formation and making conjectures and not for algorithms.

6.2.2. Concepts

The biggest problem with the knowledge base is the same problem that the heuristics have; more knowledge is needed. To aid discovery we should augment the initial knowledge base.

In addition, extra knowledge will teach us more about discovery in this domain. Currently the system tells us little about LISP's utility in data structures discovery; perhaps a logic programming language is better suited. We cannot make such conclusions based on a small program. EURISKO taught us that a close semantic-to-syntactic link is necessary in automated discovery, but the specialized nature of the ***add test step*** and the ***divide domain*** heuristics hint that LISP is not a good language to use here.

Furthermore, the knowledge base contains information that is irrelevant to the area of research. The predicates ***oddp, alphaless,*** and ***atom,*** for example, do not add anything to the discoveries. Parallel discoveries were made with structures containing numbers and with structures containing odd numbers or atoms. Unless the lexical structure of the elements is important (if we are studying hashing, for instance), the extra relations are useless and slow down the discovery process.

6.2.3. Interestingness

The interestingness calculation is severely limited. Aref uses a simple formula to decide on the worth of each new task and concept. The only measure to judge the new concept on its own merit is a test to see whether any examples can be found. If not, the concept is discarded. The weakness of this method is obvious by looking at the results. The worth of the best discovery, ***binary search,*** was 56, whereas the operation it was created from, ***search by testing*** (which is about equivalent to the original simple search), was worth 69. The system was designed so that new concepts have a lower value than the concepts they are created from; this avoids the problem of entering an infinite loop. But, although not a trivial addition, the program should have the ability of rating binary search higher.

Further evidence of the deficiency is apparent by the fine tuning that had to be done to the original worth values. The *specialize* and *generalize* heuristics, in particular, were given low values to start. Test runs demonstrated that the structures created by those heuristics, such as *list of 3 numbers,* were shallow. In contrast, AM was still able to do research when its starting concepts were given the same value. AM's heuristics for rating concepts were more useful. We need something similar in this system. The ability to measure a program's complexity would be a good start.

6.3. Summary

Aref's system has shortcomings that cannot be corrected simply. Many techniques used by AM are absent in this system although its basic structure is based on AM. More research is needed to find a means of adding these capacities to this system. The heuristics, starting knowledge, knowledge representation, and interestingness calculations need enhancement. Although the system is too small to give us good directions, it does point out several important areas that need to be tackled, and it warns us that results will not come simply.

7. Conclusions and Recommendations

Insight, illumination, and incubation are words that spell trouble for any AI system that hopes to perform a cognitive activity. Especially with discovery learning, the three I's seems to hide a clear understanding of the human ability. The hard-to-understand nature of the process creates a great challenge for AI researchers.

On the other hand, computers have advantages that can be exploited. The great speed of a computer can perhaps overcome inferiority in terms of expert knowledge. But some knowledge is needed. If we agree that our goal is to discover new-to-mankind useful knowledge, how shall we accomplish it?

The weaknesses of the aforementioned systems are apparent. Should we attempt an approach wholly different from AM's modified production system, or are AM's limitations not inherent in the general proposal? Gestalt psychology suggests that the whole of the human discovery process cannot be accomplished by AM's one-small-step-at-a-time approach; rather, human cognition is greater than the sum of its parts. On the other hand, there is no ostensible limits to

AM's method because the activity of each step is not well defined; virtually anything can be produced by an arbitrary set of complex production rules. EURISKO's successes in VLSI demonstrate the possibility.

Philosophical explanations often cite aspects of discovery that seem to be beyond the scope of AM's method. Blackwell[3] describes the epistemological and integrational aspects of discovery. All discovery implies some epistemological commitment of the researcher that allows him to interpret his results, and once any discovery is made, the results must be integrated into the overall structure of science. Although such aspects are crucial to discovery, it seems premature to consider incorporating them into the code of a system. Presumably, the production-system approach contains an implicit epistemology, and its discoveries can at least be incorporated into the overall structure of its own knowledge. It is left to the user to integrate any discoveries with the rest of science.

Instead of looking for other general approaches, or any answers to philosophical problems, we consider the current systems and suggest the important areas of future research that exist. A system such as EURISKO, with its meta-heuristics and domain dependent knowledge, provides hope for success. Perhaps BACON-like and other related systems can be incorporated into the overall system, to combine in synergy to reveal new knowledge.

7.1. Heuristic Knowledge

Whatever other limitations a discovery system may have, they seem to pale in comparison with the problem of acquiring good heuristics. Unfortunately, heuristics rarely offer any guarantee of discoveries. Since heuristics are usually compiled hindsight, they only prove to us their success in retrospect; we must hope for continuous success but cannot prove it will come. Moreover, heuristics that are newly synthesized and have no history are even less able to guarantee results. Such heuristics are commonly created by meta-heuristics that had success in the past, but future success is not guaranteed.

Nevertheless, as Lenat explains [27], it is a good meta-heuristic to assume that heuristics retain some power in the future and in slightly different domains. If we cannot suppose some regularity in the world to support this meta-heuristic, we have no recourse at all but random research. Heuristics contain knowledge about the world; in that knowledge lies the necessary power.

But which heuristics should we use, or how can we find them? There are many types of heuristics to consider. General, special-purpose, and analogous heuristics are important. The most general heuristics are the weak methods, such as generate-and-test and hill climbing, that seem to work everywhere. These methods can be specialized when more is known about the domain. If we were solving differential equations, for example, specialized techniques are necessary. Analogy comes into play when a heuristic that is successful in one situation is used elsewhere. *Study extreme examples* seems to work in number theory; it might also help in studying data structures.

Some heuristics build new concepts; others look for patterns. Some are good meta-heuristics, whereas some work best on the object level. Some are plausible space generators; others prune the space. More research is essential to provide understanding of the interaction among these sundry heuristics. One way to do this is to build new discovery systems or to expand on those already existing; another is to study the history of science or to have scientists introspect about their work. In any event, the work is fundamental to the problem of discovery.

7.2. Knowledge Representation

A second crucially important area of research is knowledge representation. As shown by AM, a proper representation greatly facilitates discovery by allowing the intended semantic change to occur by manipulating code. Representation is needed for the object level structures, concepts, and heuristics.

Pertaining to the object level, we must develop a representation with the specific domain in consideration. As each area of study has its own nuances and applicable types of production rules, the representation ought to be tailored to the domain. Even the heuristics in other domains may benefit by a unique representation since they also have domain-dependent nuances. RLL is a good step since it at least allows heuristics to have different slots in different domains, but perhaps a more flexible representation is needed to capture hard-to-formalize intuitive knowledge.

A similar problem arises when looking for an adequate representation as when acquiring heuristics: it is hard to assess the value of a solution until it has been tested by a program. When Lenat studied VLSI design, he tried several abstractions of electrical theory, each having its own representation. Although his choice helped produce discoveries, he admits that better representations may exist [38].

7.3. Domain

In [29], Lenat listed eight conclusions about mechanizing the process of discovery; seven of them concerned the choice of domain. We summarize those conclusions here.

First, the search space should be immense so that no other methods can work. If a space is too small, other common techniques, perhaps without the use of heuristics, can search a space to find objects of interest. Moreover, humans will be able to search small spaces themselves, aided by their breadth of heuristic knowledge. A computer can best exercise its power and speed in a large space. Other shortcomings may be overcome where speed is more important.

Second, the domain should be largely unexplored; the advantage of an unexplored domain is manifested by EURISKO's study of 3 dimensional VLSI design as opposed to its study of mathematics. A program has a much greater chance of proving its utility in a new area. There is no danger of being accused of cheating if new knowledge is uncovered; such knowledge seems more likely to be found where fewer discoveries have already been made.

Third, the heuristic structure ought to be complex. If many varied rules of thumb can be found that apply to a domain, there is a better chance of fruitful combinations of heuristics leading to discoveries. All types, pruners and generators, specific and general, are welcome to aid in the search.

Fourth, there must be a way to carry out experiments directly. The drawback of BACON is that it cannot do this, but rather it relies on empirical data input; therefore, it provides minimal help in the discovery process. If a system can develop theories and test them, the speed of discovery will be improved. One solution for BACON is to give it a formalization of physical properties of the world that it can use to test its theories. The obvious drawback is that the discoveries could never go beyond the formalizations. It could not propose a theory of relativity if it embodied a Newtonian formalization.

Perhaps other domain considerations exist. The domain of AI itself seems to satisfy each of the above criteria, yet it appears to be an especially poor domain. Because of the complexity of ideas in AI and the wealth of knowledge necessary to do research, AI is currently an infeasible domain for this type of work.

7.4. General Knowledge

The preceding comment suggests a barrier inherent in all automated discovery: a great breadth of knowledge is often necessary. Already in AM, by the weakness of the criteria for evaluating concepts, this barrier is encountered. A greater knowledge of the world and of mathematical aesthetics would help, but the required knowledge may be vast.

The lack of broad general knowledge is also demonstrated by the inability of a program to choose a proper representation for its concepts and heuristics. A human is required to decide on the important features that need to be coded and how to code them. It is the broad knowledge of the human in comparison to any program that precludes the computer's independence.

7.5. Interaction and Communication

There are two exterior agents that a program could conceivably interact with: the physical world and a human. If a program is researching an unformalized area as BACON does, one way to allow the program to obtain physical data is to give it the necessary interface with the outside world. But, the idea is infeasible for many domains. For instance, how could a computer or a robot carry out chemistry experiments? For now, the preferred way to retrieve data is to let a human supply it.

As described earlier, there are many other possibilities for human interaction. A human could evaluate concepts, provide goals, and suggest heuristics and meta-heuristics. This is what Lenat did to win the Naval Fleet design game. His input proved to be an irreplaceable contribution. Such interaction should be facilitated by the design of the system; then a human could alleviate the weaknesses of the computer's concept-rating and heuristic-construction.

7.6. Goal Direction and Control Structure

As suggested earlier, a goal directed search for discovery will more accurately simulate human research and therefore be more intelligible; furthermore, it may lead to discovery faster. AM had some ability to focus on related topics, but there was no notion of a general type of problem that was being investigated. EURISKO, in its attempt to design naval fleets, was in a sense goal directed; the goal was to build a fleet that would beat its current one.

This notion of goal direction is fuzzy. For example, EURISKO/VLSI was goal directed in the sense that its goal was to find designs that are cost-effective. On the other hand, it was not goal directed, because the functions of the designs were not pre-specified; any logical function was acceptable. Likewise, each domain will require its own definition of goal-direction.

There are several other considerations dealing with the control structure. For instance, it should facilitate task interaction. The success of one task ought to lead to the study of related tasks that previously were dormant and deemed uninteresting, whereas failure should lead to pruning related paths that have lost their purpose. The control structure needs heuristics to decide which tasks will lead to a solution within the present goal, and to decide which tasks among the eligible to execute first. Adding this ability is not trivial.

7.7. Resources

Production-system discovery programs use a phenomenal amount of memory and processor time. There seems to be no way around this. Expert human research also requires immense resources. Computers, because of their weaker ability to prune the space of discoveries, must spend even more time. Although no amount of processor time can overcome combinatorial problems, large resources are needed if any success is desired. Parallel processing can be used to advantage too; EURISKO currently is operating on several computers in its quest to find new heuristics in various domains.

7.8. Domain Dependent Considerations.

The current discovery systems show a wide variety of domain specific problems. For example, one deficiency of AM is theorem proving. Other areas that need work are instantiating LISP definitions and automatic programming. Aref's system requires the ability to measure the complexity of programs and also needs automatic programming techniques. Each domain that is to be automated may require a substantial amount of work before progress can be made. Not only must the user decide on the representation, but these other problems, peculiar to the domain, must be handled. This will usually entail developing a corpus of more specialized limited heuristics.

7.9. Conclusions

A wide variety of technical problems with automating discovery have been uncovered by the current related software; the most crucial being the need for heuristics. To improve the systems, heuristics must be derived from experts or synthesized by computer. Of all the programs, it seems that EURISKO in the domain of three dimensional VLSI design is the best to emulate. Much of its strength lies in its ability to manufacture heuristics, and to discover in a new, previously unresearched domain.

Unresearched domains are hard to find, but, regardless of the domain, we ought at least to supply the requisite knowledge to put our program at the frontier, so that its investigation can reach unexplored areas. We should no longer restrict ourselves to attempting rediscovery by using simple concepts and heuristics. Rather, we should allow any necessary, specific knowledge and special purpose, ad hoc procedures and unlimited interaction with human experts, in an attempt to truly discover knowledge. This is a difficult task, but it offers several benefits. No longer will there be accusations that the design of the program begs the question by containing the discoveries in its starting heuristics. When rediscoveries are made, little is proven about the viability of methods and heuristics, but true useful discoveries speak for themselves. Not only would science benefit by whatever discoveries are made, but even if no discoveries are made, we will have a much clearer picture of the research that needs to be done. A program that is designed as an honest attempt at new discoveries, even if it fails, will provide this picture and will be a great boost to the development of improved discovery systems.

8. References

[1] Aref, M. 'Automated Concept Discovery in Data Structure Applications,' *M. Sc. Thesis,* University of Saskatchewan, Saskatoon, 1984.

[2] Aref, M., 'Experiments in the Automatic Discovery of Declarative and Procedural Data Structure Concepts,' *Proceedings CSCSI/SCEIO Conference,* pp. 79-86, 1984.

[3] Blackwell, R. J., *Discovery in the Physical Sciences,* Univ. of Notre Dame Press, London, 1969.

[4] Bradshaw, G. L., P. Langley, H. A. Simon, 'Bacon.4: The Discovery of Intrinsic Properties,' *Proceeding CSCSI,* pp. 19-25, 1980.

[5] Bradshaw, G. L., P. Langley, H. A. Simon, 'Bacon.5: The Discovery of Conservation Laws,' *IJCAI* vol. 7, pp. 121-126, 1981.

[6] Brown, J. S., D. B. Lenat, 'Why AM and Eurisko Appear to Work,' *Proceedings AAAI,* pp. 246-240, 1984.

[7] Brown, J. S., D. B. Lenat, 'Why AM and Eurisko Appear to Work,' *Artificial Intelligence,* vol. 24, pp. 269-294, 1984.

[8] Bruner, J. S., *The Relevance of Education,* W. W. Norton & Company Inc., New York, 1974.

[9] Carbonell, J. G., R. S. Mickalski, T. M. Mitchell, *Machine Learning: An Artificial Intelligence Approach,* Tioga Press, Palo Alto, CA., 1984.

[10] Carbonell, J. G., R. S. Mickalski, T. M. Mitchell, *Machine Learning Part I: A Historical and Methodological Analysis,* Technical Report, Carnegie-Mellon University, May, 1984.

[11] Cohen, P. R., E. A. Feigenbaum, *Handbook of Artificial Intelligence,* vol. 4, Kaufman, Los Altos, 1982.

[12] Copeland, R. W., *How Children Learn Mathematics,* The MacMillan Company, London, 1970.

[13] Dankel, D. D., 'Browsing in Large Data Bases', *IJCAI,* vol. 6, pp. 188-190, 1979.

[14] Davis, R., D. B. Lenat, *Knowledge-Based Systems in AI,* McGraw-Hill, 1982.

[15] Ghiselin, B. (ed), *The Creative Process,* Berkeley, University of California Press, 1952.

[16] Greiner, R. D., D. B. Lenat, 'RLL: A Representation Language Language', *Proceedings First Annual Meeting of AAAI,* Stanford, CA, 1980.

[17] Hadamard, J., *The Psychology of Invention in the Mathematical Field,* Princeton University Press, New Jersey, 1949.

[18] Hanson, N. R., *Patterns of Discovery: An Inquiry into the Conceptual Foundations of Science,* Cambridge, Cambridge University Press, 1958.

[19] Hofstadter, D. R., *Godel, Escher, Bach: An Eternal Golden Braid;* Random House, New York, 1980.

[20] Koestler, A., *The Act of Creation,* Macmillan Co., New York, 1964.

[21] Kuhn,T. S., *The Structure of Scientific Revolutions,* University of Chicago Press, Chicago, 1962.

[22] Langley, P., 'Rediscovering Physics with BACON.4', *IJCAI* vol. 6, pp. 505-507, 1979.

[23] Langley, P., 'Bacon.1: A General Discovery System', *Proceeding CSCSI,* pp. 174-180, 1978.

[24] Lenat, D. B., 'Automated Theory Formation in Mathematics', *IJCAI,* vol. 5, pp. 843-842, 1977.

[25] Lenat, D. B., 'The Ubiquity of Discovery', *IJCAI,* vol. 5, pp. 1094-1105, 1977.

[26] Lenat, D. B., 'Heuretics: Theoretical and Experimental Study of Heuristic Rules', *Proceedings AAAI,* pp. 159-164, 1982.

[27] Lenat, D. B., 'The Nature of Heuristics', *Artificial Intelligence,* vol. 19(2), pp. 189-249, October, 1982.

[28] Lenat, D. B., 'Theory Formation by Heuristic Search, The Nature of Heuristics II: Background and Examples', *Artificial Intelligence,* vol. 21(1,2), pp. 41-59, March, 1984.

[29] Lenat, D. B., 'EURISKO: A Program That Learns New Heuristics and Domain Concepts, The Nature of Heuristics III: Program Design and Results', *Artificial Intelligence,* vol. 21(1,2), pp. 61-98, March, 1984.

[30] McDermott, J., D. B. Lenat, 'Less Than General Production System Architectures,' *IJCAI,* vol. 5, pp. 928-942, 1977.

[31] Poincaré, H., *The Foundations of Science,* The Science Press, New York, 1929.

[32] Polya, G., *How To Solve It,* Anchor Books, Garden City, New York, 1957.

[33] Polya, G., *Mathematical Discovery,* John Wiley & Sons, New York, 1962.

[34] Popper, K. R., *The Logic of Scientific Discovery,* New York, Basic

Books, 1959.

[35] Rich, E., *Artificial Intelligence,* McGraw-Hill, 1984.

[36] Ritchie, G. D., F. K. Hanna, 'AM: A Case Study in AI Methodology', *Artificial Intelligence,* vol. 24, pp. 249-268, 1984.

[37] Simon, H. A., 'Artificial Intelligence Research Strategies in the Light of AI Models of Scientific Discovery', *IJCAI,* vol. 6, pp. 1086-1094, 1979.

[38] Sutherland, W. R., J. Gibbons, D. B. Lenat, 'Heuristic Search for New Microcircuit Structure: An Application of AI,' *AI Magazine,* vol. 4, no. 4, September, 1982.

[39] Wilensky, R., *LispCraft,* W. W. Norton & Co., New York, NY., 1984.