BROWN UNIVERSITY
Department of Computer Science
Master's Project
CS-89-M1

"Learning Through Exploration"

by
David S. Mead

Learning Through Exploration

submitted by

David S. Mead

in partial fulfillment of the requirements for the
Master of Science Degree
in Computer Science at Brown University

January, 1989

Eugene Charniak

Eugene Charniak,*advisor*

Abstract

This project has been an implementation of learning through exploration in the field of elementary set mathematics. Using heuristics given to it, the computer is able to enhance the knowledge it has about the concepts it knows about and also to supplement those concepts with new concepts that it discovers. Discussed in this paper will be the project SiAM which does discovery in the field of set theoretic mathematics. Previously, all work on this topic has been done by Douglas B. Lenat on AM and Eurisko. These two projects will be discussed as regards their implementations, whether they are good or bad. Also discussed will be the reasons for supporting current directions of research in this area. Finally, future directions for this work are suggested.

Introduction

When considering the impressive Artificial Intelligence learning programs and paradigms, one particular group stands out as quite revolutionary. The paradigm is known as learning through exploration. The programs are AM and Eurisko. A program that learns through exploration is a heuristic search program, but it is different from most heuristic search programs. Most heuristic search programs have a well defined search space with heuristics to prune the search to the most likely nodes in that space to yield the required results. A program that learns through exploration, on the other hand, does not have a well defined search space. Thus its heuristics are used to suggest plausible moves, for without the plausible move generation, the exploration would not go anywhere. "In the first kind of search, removing a heuristic widens the search space; in AM's kind of search, removing a heuristic reduces it"([4] p.10).

The paradigm will be explored with respect to AM and Eurisko. Many questions naturally arise from such works: What problems exist within AM? What problems exist within Eurisko? Where do the benefits of such a system lie? Why are the implementations good or bad? How can the paradigm be improved and how can the implementations be improved?

In order to answer these questions, it was necessary to write a new implementation of the learning through exploration paradigm. This work is based upon the AM project done by Lenat. The description of what was done and the results are also herein.

The Program

The name of the program is SiAM (pronounced sigh-am) which stands for Simple Automated Mathematician. The program is a discovery program, that is, its "goal" is to discover new concepts. The idea behind the program is that when a human scientist sets about the task of discovering new concepts and ideas, he or she will use general rules of thumb that he or she knows about. These rules generally mutate and add to the concepts that the scientist already knows. Such is the case with a discovery program such as SiAM. The field in which SiAM does its discovery is elementary set mathematics.

Initially SiAM begins with a small number of concepts that it knows about. Examples of these concepts are Sets, Bags, Set-Union, Coalesce, Compose, etc. Concepts themselves are made up of slots which help define the concept for SiAM. There are slots such as Name, Generalizations, Examples, etc. Thus, the Sets concept would have "Set" as a value in the Name slot and it might have "{a b c}" as a value in the Example slot. At the outset of the program, many of these slots are empty. There is no information in them. SiAM's job, therefore, is to fill in this missing information from the multitude of slots that stand empty in the many concepts. As each slot has information placed into it, it is obvious that the machine "knows" more than it did before the information was filled in. Whether or not that information is useful is not yet determined, but it does know more. The machine uses heuristics, or rules of thumb to fill in that missing information. Heuristics are the backbone of the system; they guide SiAM in the discovery process. For instance, suppose SiAM wants to fill in examples of Sets. The program has a heuristic to tell it how to do this. It basically says: In order to fill in examples of a structure, find operations whose ranges are that structure and then run the operations on random examples from their domains (domain being the input value and range being the output value). Since sets are a type of structure and the current task is to find examples of sets, this heuristic executes.

These heuristics are used for basically five purposes:

1. to fill in concept slots,
2. to check the correctness of filled in slots,
3. to apply a concept's algorithm to another concept
4. to create new concepts, and
5. to suggest what new tasks can be performed.

It is the last of these purposes that gives SiAM things to do. Without suggestions to direct the program, the discovery process would go nowhere or, equally bad, the discovery process would become completely arbitrary and therefore could not be classified as a discovery process at all.

The general algorithm of the program is simply one giant loop of SiAM repeatedly getting tasks from the agenda and executing them. The agenda is simply a long list of things for the program to do. For example, one task on the agenda could be: Find examples of Set-Union. SiAM simply picks the first task off of the agenda and then finds all of the heuristics that deal with the task that has been picked. These heuristics are then executed to try to do whatever task has been proposed. After each heuristic has been executed control returns to the top of the loop so that SiAM can pick the next task to be accomplished.

The Agenda

SiAM's list of tasks that it must do is called the agenda. Tasks on the agenda have a specific simple form. The first item in a task is the action that the program should do. There are three actions, namely Find, Check, and Apply. There are two possibilities for the second item in a task. If the action is Find or Check then the second item will be the name of the slot that the action is being performed upon. If the action is Apply then the second item will specify the name of the concept whose algorithm will be applied. The third item in a task is the concept being acted upon. It is the concept whose slot is being filled (the Find action) or checked or the concept which is having an algorithm applied to it. This very simple task representation is enough for SiAM to discover several concepts and to fill in the slots of many more concepts. The entire entry for the task on the agenda has the task itself (listed just above), the reasons why this task should be performed, and the overall worth of the task based upon the worth of each of the reasons. When SiAM picks a task, it removes it from the agenda and puts it aside as something to do right now. After it finishes dealing with that task, SiAM throws it away and gets the next task from the agenda.

The tasks on the agenda are kept in order by SiAM from top to bottom in decreasing order based upon the worth of the task. The agenda is by no means stagnant. Many times a task which is already on the agenda will be suggested again, but this time for a new reason. The reason will have some value and the existing task will then have the new reason tacked on to its list of reasons and the worth of the task will be increased to reflect the addition of the new reason. Then this task must be removed from the agenda and put into its new proper place on the agenda to keep the agenda in sorted order.

For an example of the agenda in action, at one point SiAM has the task "Find Examples of Set-Union". The heuristic discussed above about finding examples of structures gets executed and fails because the domain of Set-Union is <Set Set> and there are no examples of sets yet. However, it suggests that finding examples of Sets would be good for the reason that it tried to find examples of Set-Union and failed because there are no examples of Sets. Previous to this suggestion, the task Find Examples of Sets was near the bottom of the agenda with only one reason: Finding examples of structures is worthwhile (which is a fairly unimportant reason). Now, though, the task Find Examples of Sets actually moves up to be first on the list because the value of the second reason was enough to make the worth of this task the highest on the entire agenda!

Representation of Knowledge

The knowledge representation of SiAM is fairly straightforward. There are many concepts stored upon a list known as CONCEPTS. Each concept consists of different slots in which to put information. The slots have names such as NAME, GENL, SPEC, EXMP, DEF, ALGO, etc. Within some of these slots, there are facets. Facets are really just a form of slot; they are sub-slots. These facets have two names. One of the names is just the name of the slot itself again. This facet is used to store the actual information that SiAM Finds or Checks. It also contains the information concerning the concept that SiAM starts off with. The other name is HEUR and it is in this facet that the heuristics dealing with that slot and that concept are stored. Thus, a blank concept appears as follows:

```
((NAME)
(GENL (GENL) (HEUR))
(SPEC)
(EXMP (EXMP) (HEUR))
(ISAS (ISAS) (HEUR))
(DEF)
(ALGO (ALGO) (HEUR))
(IN-DOMAIN-OF)
(IN-RANGE-OF)
(DOMAIN-RANGE)
(VIEWS)
(CONJEC (CONJEC) (HEUR))
(MYCREATOR))
```

As can be seen, not all slots have facets, namely because not all slots have heuristics attached to fill them in. This is because many of these slots become filled in as side effects of performing some other operation. For example, when applying the coalesce algorithm to Set-Union, the new concept Set-Union-Itself is created. As a result of the coalesce algorithm, Set-Union-Itself is listed as a specialization of Set-Union even though there are no explicit heuristics listed for filling in that slot. Spec, therefore, currently has no need for a HEUR facet. For consistency, and ease of alteration, of course all slots should have the two facets, but it was deemed easier for this project to leave it in the simpler form.

The information contained in each of the slots is as follows:

Name: The name of the concept
Genl: Concepts which are more general than this one
Spec: Concepts which are more specific than this one
Exmp: Things which are examples of this concept
Isas: Things of which this concept is an example
Def: Lisp definition of this concept
Algo: Lisp algorithm of this concept (Active concepts only)
In-domain-of: Actives which have this concept as part of domain
In-range-of: Actives which have this concept as their range
Domain-range: Concepts making up domain and range of this concept (Active concepts only)
Views: Lisp functions for viewing an example of one concept as an example of another
Conjec: All of the conjectures SiAM comes up with for this concept
Mycreator: The creator of this concept (perhaps a heuristic, an operation, or myself: DSM)

Although all concepts have all of these slots, not all of the slots have meaning for all concepts. Two of them, Algo and Domain-range, only apply to concepts which are active, i.e. which are examples of operations or predicates.

Heuristics

The heuristics are rules which SiAM uses to fill in slots, check slots, suggest new tasks, create new concepts, and to apply concept algorithms. The program begins with __ heuristics and all of them are found at the most general place where they are applicable. Find, Check, and Apply heuristics are attached to the HEUR facet of the various slots upon which they would be working. Suggestion heuristics are found on a separate list waiting to be executed when the agenda is empty. Heuristics are simply rules of thumb that are not necessarily always correct. If it is December and one is in Providence, one needs a winter coat to keep warm is an example of a heuristic that a person might use quite often. It is not, however, always correct. It could, for some reason, be very warm one day in December and one would not need a coat. Most of the time it is correct. This is the case with SiAM's heuristics. Usually they are correct, but not always. One heuristic in particular that is sometimes wrong is one that states that if one is trying to find examples of an operation then one can take an example of each domain element and run the operation's algorithm on them to obtain a result. These domain elements combined with the result constitute an example of the operation. Suppose then that we are trying to find examples of the operation restrict. The domain is simply Operations. Thus we can pick a random example of the concept Operation which could be Set-Union. Set-Union has Set Set as its domain. In order to restrict Set-Union, Set needs to have a specialization. It doesn't. Thus, the operation Restrict performed upon Set-Union is undefined. Restrict must return some result, so in SiAM's algorithm it returns NIL. Restrict(Set-Union) = NIL is not an example of Restrict so we see that the heuristic has failed to produce a necessarily correct answer. Usually the heuristic works, but sometimes it does not.

We could become more specific with our heuristics so that they more narrowly define their domain of applicability to reduce their rate of fallibility to zero. Suppose we more clearly specify our earlier winter example. We can add the clause "and it is cold outside" to our if clause so that it more accurately reflects the domain of applicability. Now suppose we stay inside and the heat is on; we do not need a winter coat to keep warm. We can again modify our original heuristic as to include "and one is outside" in the if clause. Now, though we need more heuristics to cover what we should wear in December in Providence.

The structure of the heuristics that find, check and apply is simple and straightforward. It is structured as and If-then construct. The first clause of the if-part decides applicability of this heuristic to the current task. It asks the following: "This heuristic applies to tasks of the form X Y Z (where X is the action, Y is a slot or concept name, and Z is a concept). Does the current task's action match X, the current task's slot or concept name match Y, and is the current task's concept a specialization or example of Z?" The subsequent clauses simply narrow down the domain a bit more by asking if miscellaneous bits of information about the concept are true. For example, the next clause might ask, "Does this concept have more than twenty examples?" The Then-part of the construct is simply the action to be taken if the If-part is satisfied. It will specify one of the five actions heuristics can perform listed above.

The heuristics which are exclusively suggestion heuristics have different structures. They do not ask about domain of applicability because they are run when the machine has an empty agenda. These heuristics simply perform the last of the five actions heuristics can do without asking any questions. These are also not stored like the rest of the heuristics, on the slot and concept to which they are applicable. They are simply kept on a separate list and all run when SiAM is out of tasks to perform. The reason for this is that unlike the rest of the heuristics, when the suggestion heuristics are run, they are all run. For example, if the action indicated by the task was Find, one would not want to run all of the heuristics that perform the action Find. On the other hand, when the action is "Suggest" (the action never really becomes Suggest, the agenda only becomes empty), SiAM should make all of the suggestions it can which means running all suggestion heuristics.

Results

SiAM's results showed much promise in what could have been acheived given more time. Beginning with set theoretic concepts, SiAM commenced by suggesting trying to find examples of the structures and operations that it knew about. While trying to find examples of some of these (such as Restrict, Coalesce, and Compose) SiAM created new concepts which needed examples filled in. After filling in examples of concepts, it suggested checking the examples and then trying to find conjectures involving those concepts. The most usual conjecture found was that the Equality predicate (a concept that SiAM knew about) held true for the domain and range, i.e. the input and output of some function were the same.

When SiAM found examples of Equality, it found that it did not have very many examples that resulted in True. Therefore, it suggested generalizing Equality, which it did, and defined Genl-Equality-One and Genl-Equality-Two which have the CAR recursion removed or the CDR recursion removed (Genl-Equality-One was really the predicate Same-Length and Genl-Equality-Two was really Same-First-Element). When SiAM canonized Same-Length with Equality, it created the concept Canonical-Bag which was really just a unary representation of numbers. The last important thing SiAM did was to use Restrict on Bag-Union to create Restricted-Bag-Union (which had its domain restricted to <Canonical-Bag Canonical-Bag>) and then to conjecture that the range of Restricted-Bag-Union was really Canonical-Bag (which it then changed in the concept Restricted-Bag-Union). Restricted-Bag-Union was really just addition.

In addition to the promising aspects, there were difficulties. For example, the concept of Compose was a difficulty by itself. Unlike the other operation generating concepts Coalesce and Restrict, Compose did not in some way reduce the concept it was working on. Coalesce changes the number of domain elements from N to N-1. Restrict changes certain domain elements to specializations of the original domain elements (and there are only so many restrictions). Compose, on the other hand, increased the number of domain elements. Also, by Composing any of the operation generating concepts with each other one created a great excess of undesirable (and many times useless) operations simply by finding examples of this operation you have now Composed.

Previous Work

There has been relatively little work done in this field, most or all of it having been done by Douglas Lenat. Lenat originally wrote a program called AM (Automated Mathematician) upon which my own work is based. It was quite similar, consisting of 115 set mathematic concepts and 243 heuristics. With all of this and AM's control structure, AM was able to discover quite alot in set theory and in number theory. AM began in set theory and slowly this evolved into number theory with the discovery of numbers. Its method of discovering numbers was using bags (multiple element, unordered structures) with only one element (in this case "T", the LISP value for true) zero or more times. Thus, AM had a unary representation for numbers. Using its heuristics, AM was able to find operations involving numbers, namely addition, subtraction, multiplication, division, squaring, etc. Finally, AM conjectured many things including the Unique Prime Factorization theorem which states that there is only one way to factor a number into prime numbers.

Lenat's next project was to begin Eurisko, a learning program that was not confined to one area of exploration. Its main purpose was to find heuristics in the domain it was currently exploring. AM was never successful at finding new, more applicable heuristics for itself, but Eurisko has been quite successful at this task. Simultaneously, Eurisko broadens its knowledge in the given field of expertise and discovers new heuristics. Eurisko's fields of exploration include battle fleet design (for a game called Traveller Trillion Credit Squadron), elementary set mathematics (like AM), and VLSI design. Working together, Lenat and Eurisko were able to design champion battle fleets two years in a row. Eurisko was able to explore set mathematics farther than AM was able to. Finally, in the field of VLSI, Eurisko discovered new three dimensional devices.

Both AM and Eurisko (as well as my own SiAM) are examples of expert systems. Usually when one refers to an expert system, systems of geological exploration or medical advice come to mind. However, AM and Eurisko are computer experts in their field and their field is simply discovery. AM is an expert Mathematics discoverer, whereas Eurisko is an expert heuristic discoverer.

Problems of AM's Approach

There were a few important problems with the approach taken when AM was created. The first problem was its inability to synthesize its own heuristics. As AM moved from set mathematics to number theory, the concepts it was working with moved farther and farther from its original set theoretic concepts. The new concepts were based upon the old concepts, but new properties also came about with the new concepts. Thus the specific heuristics that AM had were not useful because they no longer referred to the concepts that AM was working with and the general heuristics that were still valid did not capture enough of the properties of the new concepts to be truly useful.

The second important problem was AM's static list of slots which its concepts could possess. Clearly, as AM's concepts became based upon one another to define new concepts, where the new concepts were expressly defined in terms of the old concepts, the amount of information contained in each slot of the new concepts increased in comparison with the old concepts. This most notably occurred in the definitions slot and the algorithms slot, where the most modification and combination of concepts occurred. As these two slots become more complex (specifically, just longer) it becomes increasingly difficult for the heuristics to notice what would be a meaningful change and to make that change after it has been discovered. As concepts become more complex, the average length and amount of information that each slot contains increases and the average productivity of the heuristics decreases, thus decreasing the overall performance of the system.

A third related problem is that the slots tended to be too general. They were not specific enough. If one splits a single slot into two or more slots where there is a solid reason why the split occurs, then the computer has that much more knowledge about the information in each slot, even before it begins to work with the information in the slots. One example of this (which was not a problem in AM) is the division of the Examples slot into several facets like Typical, Boundary, Boundary-non-examples, and Non-examples. The program knows more about each of facets and therefore about the whole concept than it would if the Examples slot were without any subdividing facets.

In [4] on page 63, Lenat states: "One important constraint on the representation is that the set of facets be fixed for all the concepts. An additional constraint is that this set of facets not grow, that it be fixed once and for all." Clearly, Lenat's important constraint ultimately spelled AM's (and SiAM's) downfall. For a human being, it might be easy to manipulate mathematical concepts with the fixed set of slots, however, a computer has an exceedingly difficult time, due to the length of the information contained in the slots and to the generality of each slot.

Language was AM's fourth limiting factor. In order for AM to continue the discovery process, AM needed to synthesize new heuristics. However, LISP may have lent itself well to expressing mathematical concepts concisely, but it did not work very well in expressing heuristics concisely. Without the benefit of conciseness, the heuristics were too bulky to be mutated with ease and every attempt was met with very little success.

The fifth problem with the AM program is that it only has a syntactic knowledge of any of the concepts that it "knows"; there is no true semantic understanding. Parallel to this notion is the idea that the program does not have a real concept of importance. It does know when things are interesting, because it has heuristic rules that tell it this. Knowing that numbers are interesting because several other interesting concepts "use" them and because they are derived in an interesting way is not the same as saying that numbers are important. What makes numbers important? The ability to use them to count is the most important reason. Counting imposes order and at least for the human beings that use the program, imposing order upon the world is extremely important. Observing the way concepts are organized in AM, this imposition of order is also "important" to AM, but it never realizes this. Nothing is ever important to the AM program, things are only interesting.

Finally, related to the fifth problem is the sixth problem. Although it does have limited interaction with a user if that user so desires, AM conducts all of its research in a vacuum. There is no interaction with the real world, and it has no way of "knowing" anything about it. One can imagine what it might be like to have grown up inside of a 10' X 10' X 10' room with no windows and no doors and trying to do the process of discovery in some arbitrary field. The discovery process could and would quickly become strange and not quite natural. Picture trying to discovery anything about morals. It would be impossible to discover anything that held any value. Although morality is not a formalizable field (at least not in a way I can see), it illustrates the point just the same. The "outside" does matter to the process of discovery, even in a fully formalizable field like mathematics. One good example of how the "outside" could have helped AM is to look at the fourth problem with AM discussed above. With knowledge of the outside, AM may have been able to successfully discover the importance of numbers.

Eurisko's Improvements

A number of the problems with the AM system were corrected in Lenat's second project Eurisko. Those pieces that were corrected dealt with enabling

Eurisko to successfully synthesize its own heuristics which it was able to do. This enabled Eurisko to change with its changing knowledge, to keep up with the new concepts that it was producing so that it could always have domain specific heuristics with which to manipulate the concepts it had. Thus, AM's first problem was solved.

In order for that problem to be solved, however, the second problem of a static list of slots had to be solved. Using a static list of slots, one could not code the heuristics as concepts because the amount of information in each slot was too large to be usefully altered. To alter a heuristic, one would have to get at the definition of the heuristic and then mutate it. If there is too much code, it is impossible to find the piece that must be mutated, let alone mutate that piece meaningfully. Thus there had to be an expandable list of slots and facets that the concepts could possess in order to satisfy the expanding need.

The third problem is also related to the first problem because it too needed solving before the first could be solved. The third problem was that the slots were too general and needed to be more broken down into specific facets. This problem was tackled and solved somewhat in Eurisko, though it is unclear as to what extent.

The fourth problem that was solved in Eurisko was the issue of language. The approach that was taken was to create a new language called RLL which stands for Representation Language Language. This is the language which Eurisko <u>depends</u> upon. Without RLL, Eurisko would not work. Eurisko is able to manipulate the parameters of RLL to change its own language and therefore its representation in small ways.

It becomes clear therefore that these last three problems are underlying problems which needed to be solved in order to solve the first problem which is the problem that is most obvious and desirable to solve. All three of the underlying problems are questions of representation. How does the representation fall short of what is needed for the machine to create new heuristics?

Why This Direction Is Correct

In [1], Saul Amarel discusses the issue of representation as regards his example of the Missionaries and Cannibals problem. As he progresses through the paper, he introduces new representations (five in all) for the problem and each time he does, the solution becomes easier and more obvious. Clearly, in the M&C problem, the proper representation is more than half of the problem. Once the representation is good the solution for the problem flows very naturally without any struggle.

The analogy to Eurisko is clear and important. Having the correct representation for the problem allow Eurisko to execute better and cleaner and it will discover many more things with greater ease. The difficulty in Eurisko's case is that there is not just one problem to solve. Each Eurisko task is a separate problem, although a simple one. However, as Eurisko acquires more knowledge, the problem space changes because now there are more concepts and more information in the slots of the concepts. Eurisko, like the M&C problem above needs to have a good representation so that solutions (discoveries) flow from it easily. In order for Eurisko to keep a good representation in a slowly evolving problem space, Eurisko needs to be able to change its representation to match the evolving needs of the system.

Overall, it should be clear that for a program to do this kind of heuristic search, the machine needs to be able to generate new heuristics. It really needs this power for all of the reasons listed above. Eurisko's search space continually changes because of the heuristics. Therefore the heuristics must evolve as well. If not, AM's problem of heuristics which do not properly capture the nuances of a particular concept because they are too general will reappear.

Eurisko: The Good and The Bad

Eurisko's several good points certainly include its improvements over AM. It is clear that an evolving representation and an ability to alter its own heuristics when necessary are both vital and therefore good points. Eurisko has other clear advantages in terms of approach.

First, the idea of learning through exploration, through heuristic search is a good methodology. It avoids giving the program a precise problem to solve, allowing the program to find its own problems and solve them. Each task that Eurisko performs is really a small research problem. The advantage of this is that in most AI problem solving approaches, the program is given a problem and then it solves it. Someone is required to conceive of the problem in the first place before the machine can be asked to solve it. Using programs like Eurisko, the machine is able to encounter problems that the researchers themselves have not even thought to ask.

Along these same lines, a program like Eurisko can be made to discover in fields which have no real experts. The program, through experimentation, can discover on its own, domain specific heuristics for the field it is expected to work in. One of the most important pieces of information that Eurisko can provide researchers in an untrodden field IS domain specific heuristics. Using an interface the program can also be guided by the researcher in its discovery process into areas that the researcher finds interesting or important.

Additionally, a machine that learns through exploration does not view anything semantically, only syntactically. Also, the machine does not have any knowledge of the outside world to give it preconceived ideas that might "color" what it does like a true researcher has. For these two reasons, programs like Eurisko are able to approach the discovery process from a different angle than a human researcher and therefore might easily make discoveries that the human researcher is unable to realize on his or her own.

Eurisko also has some bad points. The first of these is the same as the last of the good points. The machine only manipulates syntax and it has no knowledge of the outside world to give it some direction. Most notably, it has no true notion of importance, similarly to AM.

Next, Eurisko has no notion of a goal. Again, this shows that the machine has no direction in the discoveries it makes except in what it finds interesting. With the inclusion of some sort of goal system, possibly as only suggestions of directions for the machine, the program might operate much more productively with less human interaction.

An important, but seemingly unresolvable, problem is that only highly formalizable fields of research can be implemented on a computer using this method of learning. Clearly, the machine would find it difficult the generate new heuristics for finding geological formations if it does not have the capacity to get outside and do digging and probing like a geologist. This is an unfortunate limitation to this method of learning.

Finally, Eurisko does not yet appear to alter its own representation as well as it should be able to. Truly, this is something which should happen as infrequently as possible, but from reportings by Lenat, the machine has only altered its representation a couple of times. Perhaps, this is all that was necessary and the changes that Lenat himself made were "extras", but those "extras" which make the program run better should be coded as more heuristics, if possible so that Eurisko can do this routine maintenance itself.

Future Directions

The future directions for a program like Eurisko which learns via heuristic search seem to be many. Some of these directions are:

1. Seek new formalizable fields in which to do discovery
2. Seek partially formalizable fields in which to aid a researcher to do discovery
3. Give Eurisko more heuristics in order to alter its own representation
4. Give Eurisko itself as a domain to do discovery and possibly discover representations for itself that would be better suited for itself.
5. Give Eurisko the domain of knowledge representations to explore.
6. Give Eurisko the ability to ask questions of the researcher using the program.
7. Try to give Eurisko "intuition". Try to give it a sense of the real world so it knows importance.
8. Try to refine Eurisko's representation as fine as possible and classify it all so that Eurisko has more information. Give Eurisko the ability to do this itself (See 3, 4 ,5)
9. Give Eurisko a goal-based system to help guide it. Perhaps the goal system can simply be something with which Eurisko can guide its own discovery process more effectively.

Conclusion

One of the most important and powerful methods for machine learning has been learning through exploration. With this technique, the computer is able to learn by itself or with the help of a researcher. A program, written in this style, can acheive great success learning and discovering in highly formalizable disciplines. It is truly a technique that has earned its mark in Artificial Intelligence and should lead the way towards better learning techniques and programs.

Bibliography

[1] Amarel, Saul, "On Representations of Problems of Reasoning About Actions", *Machine Intelligence 3*, pp 131-171, (1968).

[2] Charniak, Eugene and McDermott, Drew, Introduction to Artificial Intelligence, pp 642-650, Addison-Wesley, Reading, 1985.

[3] Lenat D. B., "On Automated Scientific Theory Formation: A Case Study Using the AM Program", *Machine Intelligence 9*, pp 251-283, (1979).

[4] Lenat, Douglas B., "AM: Discovery in Mathematics As Heuristic Search", pp 1-225 in Knowledge-Based Systems in Artificial Intelligence, ed. Randall Davis and Douglas B. Lenat, McGraw-Hill, New York, (1982).

[5] Lenat, Douglas B., "The Ubiquity of Discovery", *Artificial Intelligence 9*, pp 257-285, (1977).

[6] Lenat, Douglas B., "Theory Formation by Heuristic Search", *Artificial Intelligence 21*, pp 31-59, (1983).

[7] Lenat, Douglas B., "Eurisko: A Program That Learns New Heuristics And Domain Concepts", *Artificial Intelligence 21*, pp 61-98, (1983).

[8] Lenat, Douglas B. and Brown, John Seely, "Why AM and Eurisko Appear to Work", *Artificial Intelligence 23*, pp 269-294, (1984).

[9] Ritchie, G. D. and Hanna, F. K., "AM: A Case Study in AI Methodology", *Artificial Intelligence 23*, pp 249-268, (1984).