Linköping Electronic Articles in
Computer and Information Science
Vol. 5(2000): nr 38

Automated Theory Formation Applied to Four Learning Tasks

Simon Colton

Linköping University Electronic Press
Linköping, Sweden

http://www.ep.liu.se/ea/cis/2000/038/

Published on December 21, 2000 by
Linköping University Electronic Press
581 83 Linköping, Sweden

Linköping Electronic Articles in
Computer and Information Science
ISSN 1401-9841
Series editor: Erik Sandewall

©2000 Simon Colton
Typeset by the author using LaTeX
Formatted using étendu *style*

Recommended citation:
<Author>. <Title>. Linköping Electronic Articles in Computer and Information Science, Vol. 5(2000): nr 38. http://www.ep.liu.se/ea/cis/2000/038/. *December 21, 2000.*

This URL will also contain a link to the author's home page.

The publishers will keep this article on-line on the Internet (or its possible replacement network in the future) for a period of 25 years from the date of publication, barring exceptional circumstances as described separately.

The on-line availability of the article implies a permanent permission for anyone to read the article on-line, to print out single copies of it, and to use it unchanged for any non-commercial research and educational purpose, including making copies for classroom use. This permission can not be revoked by subsequent transfers of copyright. All other uses of the article are conditional on the consent of the copyright owner.

The publication of the article on the date stated above included also the production of a limited number of copies on paper, which were archived in Swedish university libraries like all other written works published in Sweden. The publisher has taken technical and administrative measures to assure that the on-line version of the article will be permanently accessible using the URL stated above, unchanged, and permanently equal to the archived printed copies at least until the expiration of the publication period.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/ *or by conventional mail to the address stated above.*

Abstract

Automated theory formation involves, amongst other things, the production of examples, concepts and statements relating the concepts. The HR program [5] has been developed to form theories in mathematical domains, by calculating examples, inventing concepts, making conjectures, and settling conjectures using the Otter theorem prover [13] and MACE model generator [14].

In addition to providing a plausible model for automated theory formation in pure mathematics, HR has been applied to other problems in Artificial Intelligence. We discuss HR's application to inducing definitions from examples, scientific discovery, problem solving and puzzle generation. For each problem, we look at how a theory formation approach can be applied and mention some initial results from the application of HR. Our aim is not to describe the applications in great detail, but rather to provide an overview of how HR is used for these problems. This will facilitate a comparison of the problems and discussion of the effectiveness of theory formation for these tasks.

Our second aim is to compare HR with the Progol machine learning program [17]. We do this first by looking at the concept formation these programs perform. Also, by suggesting how Progol could be used for the applications mentioned above, we compare the programs in terms of how they can be applied.

Author's address

Division of Informatics
University of Edinburgh
Edinburgh EH1 1HN
United Kingdom

E-mail: `simonco@dai.ed.ac.uk`

1 Introduction

A theory often discusses objects of a particular nature. For example, in pure mathematics, number theory is about integers, whereas graph theory concerns graphs and group theory concerns groups. Similarly, in non-mathematical domains there are objects of interest around which a theory forms, for example acids in chemistry, sub-atomic particles in physics and so on. Theories typically contain (i) examples of the objects of interest, (ii) concepts which discuss the nature of those examples and (iii) statements highlighting relationships between concepts. For example, in finite group theory, there are 14 groups up to isomorphism with 8 or fewer elements. There are also many concepts describing these groups, for example cyclic groups are a particular type of group and the centre of a group is a subset of elements of the group. Group theory also contains many statements relating two or more concepts, for instance if a group is cyclic, then the centre of the group will contain all the elements, i.e. it will be Abelian. Similarly, in chemistry, there are examples of acids, such as hydrochloric and there are specialisations of the concept of acids, for instance organic and inorganic acids. There are also statements about acids, such as: adding an acid to a base will produce a salt and water.

In mathematics, the statements are often *proved* via a sequence of logical inferences. The statements are usually called conjectures until they are proved, when they become theorems. Theories will contain proofs, disproofs and counterexamples as well as open conjectures for which the truth is unknown. In non-mathematical domains it is often possible to formalise the statements and appeal to mathematical proofs. However, sometimes the plausibility of a statement has to be *demonstrated* with experiments and *explained* via more theory formation. For instance, experiments where acids and bases are mixed add plausibility to the above statement, because a salt solution is usually observed. To explain this phenomena, chemists may provide a reaction mechanism to show how the bonds in the chemicals break and re-form during the reaction.

Given this initial synopsis of what theories contain, automated theory formation should be able to at least find examples of the objects of interest, invent new concepts and make plausible statements relating those concepts. In mathematics, theory formation should also involve proving and disproving conjectures. There have been many automatic approaches to these individual tasks. For instance, the Progol program [17] can invent new concepts and the MECHEM program [23] can find reaction pathways in chemistry. Similarly in mathematics, the Mathematica program, [24] can perform calculations and symbolic manipulations, the AGX and Graffiti programs [1], [10] can make conjectures, the Otter program [13] can prove conjectures and the MACE program [14] can find counterexamples.

There have only been a few attempts to automate theory formation as a whole. The AM program [12] was the first to explore mathematical domains using concept formation and conjecture making. The GT program [9] automated more mathematical activities by enabling example generation and theorem proving as well as concept formation and conjecture making. The HR program [5] performs automated theory formation in domains of pure mathematics. Using all of its functionality, HR can start with just the axioms of a finite algebra such as group theory. It will then find examples, invent concepts, make conjectures, prove theorems and find counterexamples to false conjectures. HR can also work in number theory and graph theory and we intend to use HR in more mathematical domains.

As well as providing a plausible model for theory formation, HR has been applied to other problems in Artificial Intelligence. In §3 we discuss four such problems, namely the induction of definitions from examples, scientific discovery, problem solving and puzzle generation. We do not aim to give a complete description of the application of HR to these problems but rather to give an overview of our approach using HR. We also suggest how the Inductive Logic Programming (ILP) program Progol [17] could be used for these tasks and we compare HR and Progol in terms of their application. However, we begin in §2 by comparing HR and Progol in terms of the concepts they form.

2 The HR and Progol Programs

2.1 The HR Program

The HR program [5], named after mathematicians Hardy and Ramanujan, is designed to form theories in domains of mathematics such as group theory, graph theory and number theory. HR starts with background information such as the axioms of a finite algebra, or some concepts in number theory such as the divisors of integers, multiplication and addition. Each concept is supplied with a definition and the user can also supply a finite set of examples, although this is not necessary in algebraic domains, as examples can be generated from the axioms. HR uses one of seven general production rules to base a new concept on either one old concept (in which case we say the production rule is *unary*) or two old concepts (a *binary* production rule). This produces a set of concepts which form the core of the theory.

Each production rule generates a definition and a set of examples for the new concept and table 1 describes the action of each production rule. For example, starting with the concept of divisors of integers in number theory, figure 1 shows how HR constructs the concept of prime numbers. This concept is produced using the size production rule to count the number of subobjects (divisors) followed by the split rule to instantiate this number to 2. This extracts those numbers with exactly two divisors — prime numbers. For a more detailed description of the production rules, see [7].

Rule	Action of Production Rule
Compose	Composes predicates by conjunction
Exists	Introduces existential quantification
Forall	Introduces universal quantification
Match	Equates variables in predicates
Negate	Finds compliments to predicates (negating the property)
Size	Counts the number of subobjects satisfying a predicate
Split	Instantiates variables

Table 1: The action of HR's seven production rules

It is important to note that a concept has (i) a set of examples, (ii) a definition, (iii) a categorisation over the examples HR has available and (iv) a set of conjectures involving the concept. For instance, if HR is working with the integers 1 to 10 in number theory, then the concept of prime numbers will have these examples: $\{2, 3, 5, 7\}$ and the definition given in figure 1. We call this a *specialisation* concept because it produces a binary

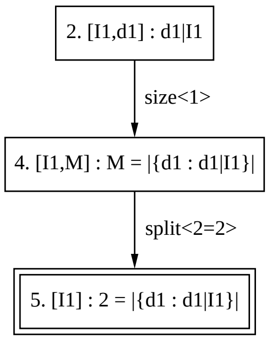


Figure 1: Construction of the concept of prime numbers

categorisation of the integers which specialises the concept of integer into prime and non-prime integers thus:

$$[1, 4, 6, 8, 9, 10], [2, 3, 5, 7]$$

In the theory HR produces, there will also be a set of conjectures about prime numbers, for example that prime numbers are never perfect squares. While producing concepts, HR makes these conjectures using empirical evidence. In particular, if it notices that the examples of a new concept are exactly the same as an old concept (for the data available), it will conjecture that the definitions of the two concepts are logically equivalent — producing an 'if and only if' conjecture. Similarly, if it notices that the examples of one concept are all examples of another concept, it will make an implication conjecture. If it cannot find any examples for a concept, it will make a non-existence conjecture (i.e. that there are no examples whatsoever). In finite algebras, HR invokes the Otter theorem prover [13] to prove the conjectures it makes. Whenever Otter is unsuccessful, HR uses the MACE model generator [14] to find a counterexample to disprove the conjecture. In this way, HR forms a theory which contains concepts, examples, open conjectures, theorems and proofs.

To improve the quality of the theories, HR uses heuristic measures to estimate the worth of concepts and performs a best first search by using the more interesting concepts as the basis for new concepts before the less interesting ones. The user sets weights for a weighted sum of all the measures which is taken as an estimate of the worth of each concept. The measures include intrinsic properties of the concept such as the number of examples it has, as well as relational measures such as the novelty of the categorisation it produces, as discussed in [5]. The quantity and quality of conjectures that a concept appears in is also assessed, with concepts appearing in interesting conjectures assessed as more interesting than those appearing in dull conjectures. The worth of a theorem is assessed by the length of the proof produced by Otter, with longer proofs indicating a more interesting conjecture statement. HR therefore completes a cycle of mathematical activity where concept formation drives conjecture making and theorem proving which in turn improves concept formation. HR improves on previous theory formation programs such as AM [12] and GT [9] by incorporating theorem proving (AM could not prove theorems) and by being able to work in many domains (GT could only work in graph theory).

```
% Mode Declarations
:- modeh(1,square(+intgr))?
:- modeb(1,multiply(+intgr,-intgr,-intgr))?

% Background Knowledge
intgr(1).intgr(2).intgr(3).intgr(4).intgr(5).
intgr(6).intgr(7).intgr(8).intgr(9).intgr(10).
multiply(A,B,C) :- intgr(B), intgr(C), A is B*C.

% Positive Examples
square(1).square(4).square(9).

% Negative Examples
:- square(2). :- square(3). :- square(5).
:- square(6). :- square(7). :- square(8). :- square(10).
```

Figure 2: Input to Progol for learning the concept of square numbers

2.2 The Progol Program

Inductive Logic Programming (ILP) is a general purpose machine learning technique [16]. Concepts are represented as first order logic programs, which has many advantages, including that they can be interpreted by an underlying logic programming language. The Progol program [17] uses ILP with an underlying Prolog interpreter. Progol is usually employed to produce a logic program which defines a set of given positive examples but not the given negative examples. The definitions are based on background predicates supplied by the user.

As an example, Progol can learn the concept of square numbers, given the background knowledge and positive and negative examples in figure 2. Progol produces this answer:

```
square(A) :- multiply(A,B,B).
```

This is a Prolog program which will identify a square number as being the multiplication of some number with itself. The mode declarations at the top of the input in figure 2 determine the format for the logic program to be learned, with `+` indicating the use of a known variable, `-` indicating the introduction of a new variable and `#` indicating possible instantiation. Progol searches for concepts using the U-Learnability framework [19]. In this framework, there is a prior probability distribution over the space of concepts, with the probability being the likelihood that the concept is the required one.

The construction of new concepts is achieved by inverting deductive rules of inference to produce inductive rules. One rule of deduction which is inverted is the resolution rule [20]. In its simplest form, this states that if we know:

$$A \rightarrow B \text{ and } B \rightarrow C$$

then we can deduce that:

$$A \rightarrow C$$

The first two ways to invert resolution involve inverting a single resolution step. In effect, this amounts to asking the question: 'given the observed clauses [logic programs] in the data, what two clauses could have been resolved together to give this observation?' The absorption and identification inductive rules of inference are obtained in this way:

Absorption: $$\frac{q \leftarrow A \qquad p \leftarrow A, B}{q \leftarrow A \qquad p \leftarrow q, B}$$

Identification: $$\frac{p \leftarrow A, B \quad p \leftarrow A, q}{q \leftarrow B \qquad p \leftarrow A, q}$$

The absorption rule can be read as: 'Given that I observe $q \leftarrow A$ and $p \leftarrow A, B$, one hypothesis I can make is that this is because $q \leftarrow A$ and $p \leftarrow q, B$ are true and have been resolved to produce the observations. By interpreting this hypothesis as a logic program, the feasibility of it being true can be checked against the data.

Two more induction rules are derived from inverting 2 resolution steps:

Intra-Construction: $$\frac{p \leftarrow A, B \qquad\qquad\qquad p \leftarrow A, C}{q \leftarrow B \qquad p \leftarrow A, q \qquad q \leftarrow C}$$

Inter-Construction: $$\frac{p \leftarrow A, B \qquad\qquad\qquad q \leftarrow A, C}{p \leftarrow r, B \qquad r \leftarrow A \qquad q \leftarrow r, C}$$

With intra-construction, the hypothesis produced states that clauses $q \leftarrow B$ and $p \leftarrow A, q$ are true and were resolved to give the observed $p \leftarrow A, B$ and clauses $p \leftarrow A, q$ and $q \leftarrow C$ were resolved to give the observed $p \leftarrow A, C$. A new predicate symbol, q, has been introduced and likewise the predicate r is introduced in the inter-construction rule. This phenomena is called **predicate invention** and is often necessary to enable ILP programs to learn the correct definition for a concept. For example, when constructing a logic program for 'insertion sort', intra-construction is required to introduce an 'insert' predicate [18].

2.3 Concept Formation in HR and Progol

There is a striking similarity between the concepts Progol and HR can form. We highlight this using examples from number theory. Firstly, in Progol, concepts are formed which have definitions with conjunctions of predicates and the predicates may have variables repeated within them and between them. This produces concepts that HR can form with its compose, match and exists production rules. For example, given the background concepts of integers and multiplication, HR produces this definition for square numbers:

$$[n] \;:\; \exists\, a \;\; (a \times a = n)$$

and Progol produces this definition:

```
square(A) :- multiply(A,B,B).
```

Secondly, in Progol, the user can set mode declarations describing where background predicates can appear in the invented predicates. Mode declarations also specify whether variables become instantiated and whether negation of predicates is allowed. The ability to instantiate variables corresponds exactly with HR's split production rule, and the ability to negate predicates corresponds with the negate rule. Also, a combination of negated and existentially quantified predicates corresponds to concepts produced by HR's forall production rule. For example, HR defines even numbers as:

$$[n] \; : \; 2|n$$

Similarly, given the background predicate of divisors and allowed to instantiate variables, Progol produces this definition:

```
even(N) :- divisor(N,2).
```

Finally, we found that if we supply two additional predicates as background knowledge from set theory, namely the standard Prolog predicates of `setof` and `length`, Progol can cover concepts produced by the size production rule. For example, HR defines the τ function (number of divisors of an integer) in this way:

$$[n,t] \; : \; t = |\{a : a|n\}|$$

and Progol produces this equivalent definition:

```
tau(N,T) :- setof(M,divisor(N,M),L),
            length(L,T).
```

Therefore, for each of HR's production rules, Progol can produce concepts of a similar nature. Interestingly, to cover all the production rules requires three different aspects of Progol's functionality, yet only one production rule corresponds to additional background knowledge. Progol has greater coverage of concepts than HR. In particular, Progol can define concepts recursively by specifying a base case and a step case. HR cannot yet produce such concepts, although we plan to implement another production rule to enable this.

3 Applications of Theory Formation

3.1 Inducing Definitions from Examples

The problem of inducing a definition for a concept given some positive examples of the concept and possibly some negative examples is well known in machine learning, and we have explored the possibility of using HR in this fashion. We have used HR to learn definitions for integer sequences, as discussed in [7] and have also applied HR to Michalski-style train problems [15] where the program is asked to find a reason why a certain subset of trains are going east, based on certain characteristics of the train, for example the shape of the carriages.

A naive way to use theory formation for learning tasks is to supply HR with background knowledge and ask it to form a theory, stopping when it has found a concept which matches the data supplied. To focus theory formation, we adapted HR's heuristic search to favour building on concepts which achieved a categorisation closer to the one achieved by the target concept. We found that this approach often failed to learn integer sequences because there was no discernible gradient for the measures HR uses, and so hill climbing was not possible (see [7] for further details).

Instead of the heuristic search, we used a 'unary first' search enhanced with a look ahead mechanism. A *unary first* search is a combination of a depth first and breadth first search: the unary production rules are used exhaustively for each new concept before returning to the binary production rules with old concepts. In this way, each new concept receives some preliminary development, but is not combined with previous concepts until later. As an example of the look-ahead mechanism, given the sequence $2, 3, 5, 7$ (prime numbers) we have enabled HR to notice that, when it forms the concept of number of divisors, these numbers all have two divisors, a fact which is true of none of the other integers in HR's dataset. Each production rule has an algorithm for noticing a pattern which is true only for the positive examples, and when this happens a suitable theory formation step involving that production rule is added to the top of the agenda. Execution of the step produces a concept which fits the data. The pattern-spotting mechanism is faster than actually performing a theory formation step because there are overheads involved in performing a theory formation step and for the majority of the time, it is possible to quickly tell that there is no patter.

The look ahead mechanism has been successful with both problems about trains and integer sequences, and we supply some results in [7]. It is particularly effective when the concept to be learned is a combination of two old concepts, e.g. the concept of odd prime numbers, which combines the concepts of odd numbers and prime numbers. Depth first, breadth first and unary first searches do not find this concept quickly without the look ahead mechanism. However, with the look ahead mechanism, odd numbers are invented and as soon as prime numbers are introduced, HR notices that the positive examples are both odd and prime (and the negative examples are not). HR then combines these concepts and reaches the solution much faster — the time taken to learn the concept reduces from 384 to just 5 seconds. For more information on HR's application to learning tasks, please see [7].

The Progol program has been specifically designed to perform such induction tasks and has had much success. No further description of how it operates is required.

3.2 Scientific Discovery

In less than an hour, HR can produce more than 2000 concepts in number theory. Hence there is the possibility of HR producing new and interesting concepts, but it is difficult to tell in general whether a concept is either new or interesting. In number theory, however, there is an Encyclopedia of Integer Sequences [22] which contains around 60,000 sequences collected over 35 years by Neil Sloane, with contributions from many mathematicians. If a concept HR produces in number theory can be interpreted as an integer sequence which is missing from the Encyclopedia, this gives some indication — but by no means a guarantee — that the concept may be novel.

We have also used the Encyclopedia to give some indication as to whether the new integer sequences HR produces are interesting. To do this for a chosen sequence S, we have enabled HR to find sequences in the Encyclopedia which are empirically related to S, with the relations interpreted as conjectures about S. As a trivial example, given the sequence of prime numbers, HR makes the conjecture that they are never square numbers. It does this by noticing that none of the prime numbers it has are in the Encyclopedia entry for square numbers. As well as finding disjoint sequences, HR is able to find subsequences and supersequences of the chosen sequences.

Due to the large number of sequences in the Encyclopedia, many sequences related to the chosen one are output and we implemented pruning techniques to discard dull results. For example, it is desirable that a sequence conjectured to be disjoint with the chosen sequence has its terms distributed over roughly the same part of the number line as the chosen sequence. If so, the two sequences occupy roughly the same part of the number line yet do not share any terms — which increases the possibility of the conjecture being true and/or interesting. Therefore, HR discards conjectures about disjoint sequences if the overlap of their ranges falls below a minimum specified by the user.

By finding conjectures relating the sequence HR has invented to the sequences already in the Encyclopedia, HR provides some evidence that the sequence is of interest. This 'invent and investigate' approach has successfully led to 20 sequences invented by HR being added to the Encyclopedia, all supplied with interesting conjectures. A good example of this is the sequence of integers where the number of divisors is prime, which HR invented (in as much as it was produced by HR and not present in the Encyclopedia). When asked to find subsequences of this sequence, the first answer produced was the sequence of integers where the *sum* of divisors is prime (submitted to the Encyclopedia by someone else). Interpreted as a conjecture, this result states that, given an integer, n, if the sum of divisors of n is prime, then the number of divisors of n will also be prime. We have subsequently proved this result, and while we do not know for certain whether it is new, it certainly adds interest to the sequence HR invented. For more information on the application of HR to the invention of integer sequences, see [2] or [8].

While HR has produced many new sequences using the invent and investigate approach, it has also produced a new sequence by finding a definition for a given sequence. That is, we determined that the Encyclopedia of Integer Sequences contained a sequence starting a, b, c, d for all a, b, c, d such that $0 < a < b < c < d < 10$ with two exceptions. There was no sequence starting $4, 5, 6, 9$ and no sequence starting $4, 5, 7, 9$. We set HR the task of inventing sequences starting with these terms. In the latter case, within seconds, HR identified that the concept of prime numbers + 2 fitted the examples and this sequence is now in the Encyclopedia. While HR found a solution for the first sequence, the definition was fairly complicated (see [7]), and so we have not submitted it to the Encyclopedia.

In general, Progol has also been used to perform scientific discovery tasks by identifying a definition for a concept for which the categorisation of the examples into positives and negatives was already known. For instance, when applied to data from experiments involving the inhibition of E. Coli Dihydrofolate Reductase [11], the positive examples of the concept were pairs of drugs d_1 and d_2, where d_1 was known to be more effective at the inhibition task than d_2. The task was then to learn a definition for this concept, in effect to find a rule describing why d_2 was less effective. Within the rule derived for the concept, there may be new concepts, but the emphasis is on finding a definition for a known concept. To our knowledge, Progol has not been used in a way similar to HR above, where an entirely new concept and/or statement was identified and shown to be interesting.

3.3 Creative Problem Solving

In his book on mathematical problem solving [25] Paul Zeitz suggests a 'plug-and-chug' method, whereby calculations are performed and the results analysed to see if a pattern emerges which might provide insight into the problem. Zeitz supplies the following problem — taken from a 1930s Hungarian mathematics contest — as an example where this approach leads to the solution:

Show that the product of four consecutive integers is never a square number.

Following the plug and chug method, Zeitz calculates examples of the product of four consecutive integers:

$$1 \times 2 \times 3 \times 4 = 24 \text{ and } 2 \times 3 \times 4 \times 5 = 120$$

The sequence of calculations continues: 24, 120, 360, 840 and a eureka moment occurs with the realisation that these are all one less than a square. Zeitz then makes the conjecture that all such numbers are one less than a square and hence not square numbers. Zeitz states that:

> 'Getting to the conjecture was the crux move. At this point the problem metamorphosed into an exercise!'

To finish the problem, it is necessary to show that the product of four consecutive integers can be written as a square minus 1:

$$n(n+1)(n+2)(n+3) = (n^2 + 3n + 1)^2 - 1.$$

We have applied HR to plug-and-chug problems of this nature, by getting it to make suggestions which might lead to a eureka moment for the user. To do this, HR is given a set of numbers which are related to the problem and asked to suggest properties of the numbers in the hope that one of the suggestions will provide an insight. To do this, for every new concept HR introduces, if all the given numbers have the property prescribed by the concept, then the definition is output. For example, when used for the Hungarian contest problem above, HR is given the numbers 24, 120, 360 and 840. As it forms a theory, it invents types of number and when the numbers 24, 120, 360 and 840 all satisfy the definition of a particular number type, the definition is output. Of course, some suggestions do not provide insight (for example that they are all even numbers). However, HR eventually invents the concept of squares-minus-one and so finds the conjecture which metamorphosed the problem.

The application of HR to problem solving is very recent and we are still experimenting and compiling a corpus of problems where the plug-and-chug approach would help. We hope to attach this functionality to a computer algebra system such as Maple or Mathematica. For more information on the application of HR to problem solving, see [4].

We have not applied Progol to this type of problem, so we can only speculate on how to do this. The problem here is not to learn a definition for a given concept, but rather to learn a *property* of a given concept. In machine learning terminology, the given concept can be thought of as a cluster and this problem is to find a larger cluster containing the given one. With HR, we chose to do this by finding new concepts which were generalisations of the given concept. One way to do this with Progol would be to include some negative examples along with the positive examples and attempt to learn a definition for this concept, which would be a generalisation of the one given. Deciding which negative examples to include would possibly be problematic and systematically choosing them may be too time consuming.

3.4 Puzzle Generation

Theorem proving has attracted much more attention than conjecture making in automated mathematics and similarly, the problem of finding solutions to puzzles [21] has been much more researched than the question of generating interesting puzzles. We are interested here in one particular type of puzzle, namely odd one out puzzles. Such puzzles ask the problem solver (assumed to be a human from here on) to choose one object out of a set of similar objects and give a reason for the choice. The reason must be in terms of a property which the others share but which is not true of the object they have chosen, hence it is the odd one out.

We formalise the problem of generating odd one out puzzles in the following way: a puzzle is a set of n objects taken from a (possibly infinite) set of examples supplied by the user and a specialisation concept which categorises them into $n-1$ positive examples and 1 negative example. The negative example is the odd one out in the solution and the concept producing the categorisation provides the reason why it is the odd one out. We will concentrate here on the case where $n = 4$. For example, given the integers 1 to 20, then the concept of even numbers and the set of integers $\{2, 10, 17, 20\}$ forms a puzzle because $2, 10$ and 20 are even, but 17 is not.

To add to our specification of the problem, we note that the solution to the puzzle must be *satisfying* to human solvers. There are many ways in which a solution could be unsatisfying, but we concentrate on only one here: if there is another solution of similar or lesser complexity than the solution given, this will be unsatisfying. As an example, consider the following puzzle:

Which number is the odd one out?

4 9 16 30

There are at least two simple solutions to this puzzle:

- 9 is the odd one out, because the others are even, yet 9 is odd
- 30 is the odd one out, as the others are square numbers but 30 is not

The first solution is perhaps most likely to be given as the answer because even numbers are more easily recognised than square numbers. However, this does not detract from the fact that the solutions are of similar complexity and if the solver gave one solution but the 'correct' one was the other, the solver would probably be dissatisfied with the puzzle. Hence an additional criteria for puzzles is that they have no other solution of similar or lesser complexity. We can use HR to increase the likelihood that a puzzle satisfies this criteria, but we do not claim to rule out other solutions completely, and any puzzle HR produces may be unsatisfying. However, the same is true of human generated odd one out puzzles.

The application of HR to puzzle generation is still in its early stages. The domain we have used so far has a finite number of examples which we call 'pgrams', a shortening of 'puzzle diagrams'. In figure 3, we give four example pgrams. Each pgram has either a circle, square or triangle in each of the four corners, so there are $3^4 = 81$ pgrams in total. The initial concepts HR starts with in this domain only describe which shapes are in which positions. HR is not yet given more complicated concepts such as diagonals or rotation and reflection of one pgram to produce another.

To produce puzzles, we start HR with just one pgram and use it to perform concept formation with all the production rules other than compose, which enables it to exhaust the search. HR introduces counterexamples to false conjectures, and because there are only 81 pgrams in total, HR searches all of them for a counterexample. The search is exhausted after 25

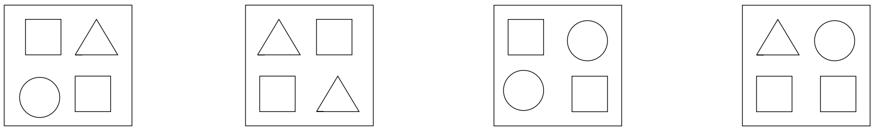
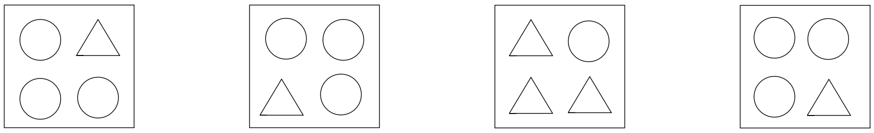


Figure 3: Four pgrams

Which is the odd one out?



1 2 3 4

Figure 4: Puzzle generated by HR

seconds by which stage 14 pgrams have been introduced as counterexamples and HR has defined 62 specialisations of pgrams. HR then takes each specialisation concept S in turn and attempts to embed it into a puzzle. To do this, HR searches for 3 positive and one negative example of S. These have to be chosen in such as way that none of the other 61 specialisations provide a rival solution. A rival solution is one for which the odd one out differs to the negative example chosen for S. Choosing examples for S for which there is no rival solution increases the chance that the puzzle will be satisfying, but does not guarantee it.

We are still experimenting with different strategies for producing puzzles and more work needs to be done to increase the yield. Using the above approach, only 5 distinct puzzles were found, including the one in figure 4. This puzzle embeds the concept of having exactly one triangle, hence the odd one out is number 3, and this puzzle is easy to solve. More importantly, however, the rival solutions to the puzzle seem to be more contrived. For instance, number 4 could be considered the odd one out because it has two circles on its bottom-left to top-right diagonal, whereas the others have both a circle and a triangle. HR did not start with the concept of diagonals or invent the concept itself, so it did not notice this rival solution. With the rival solutions being more contrived, it seems likely that, while it is easy to solve, the solution to this puzzle will be satisfying to most people, although we need to confirm this with further experimentation.

Again, we can only speculate on the use of Progol for this application. The learning task we set HR was to produce a set of specialisation concepts which had good coverage of the simple concepts in a domain, so that rival solutions can be checked. By giving Progol many different binary categorisations of the pgrams, it could learn definitions for many concepts, keeping those which are below some pre-defined complexity limit. However, there are far too many ways to categorise the 81 pgrams, so either some selection of the categorisations would be required, or a smaller number of pgrams could be used. For example, there are around 10,000 different ways to categorise 14 pgrams into positive and negative examples, and it may be possible to learn definitions for this set.

Perhaps a more feasible alternative use of Progol for this task would be the following. Firstly, choose 4 pgrams from the set of 81 and choose one of these to be a negative example, with the other three being positive examples. Then attempt to learn a concept with this categorisation of the examples and record the complexity of any definition produced. If this is achieved, a legal puzzle will have been generated and it will be necessary to check for rival solutions. One way to do this would be to re-categorise the four examples, choosing a new negative example and attempt to find a new definition. If only definitions with much larger complexity than the first one could be found, the puzzle will have no simple rival solution. This approach appears to be as plausible as our approach with HR, although we need to experiment to check this. However, the problem with this approach might the small number of examples: only three positive and four negative examples. With such a small number of examples, Progol may not be able to learn a definition which achieves any compression.

4 A Comparison of the Four Applications

By highlighting some commonalities between the four applications described above, we can draw some conclusions about the application of theory formation in general.

Our first observation is that with all four applications, part of the goal is to learn a concept which has certain properties. This is clear with the application to inducing a definition from examples, where the goal is to find a concept which achieves a given categorisation of the examples supplied. With scientific discovery — in the way that HR performs it — the goal is to find a concept for which even the categorisation is not known. The concept must have the property of being interesting. With the application to creative problem solving, the aim is to find a concept which is a generalisation of the given concept. With the application to puzzle generation, the aim is to find a concept for which examples can be found for a puzzle, for which there is no simple rival solution. To check that there are no rivals, HR also needs to generate a large set of concepts from which a rival might be found.

Hence we can conclude that three main applications of theory formation are (i) to find something about a given concept (i.e. a definition, or a property), (ii) to find an entirely new concept with a particular property and (iii) to find a set of concepts which cover all definitions of a particular form. With the exception of the generation of novel integer sequences, concept formation has been the main aspect of theory formation required for the problem. However, as we discuss in §5, we also hope to apply the conjecture making aspects of theory formation to areas of Artificial Intelligence, in particular constraint satisfaction problems and automated theorem proving.

The role of the user differs between each task. The user takes no part in the puzzle generation or the application to induction of definitions (other than supplying the positive and negative examples and perhaps making some adjustments to the settings). However, in the application to creative problem solving, the user must interpret the property HR suggests and determine whether it provides insight to the problem at hand. Similarly, with the discovery of integer sequences, the user must interpret the relations HR finds as conjectures and attempt to prove or disprove them. In this case, the user also has to choose one of HR's new sequences to investigate.

For each application, HR performed a different search for concepts and was enhanced with an additional module to complete the task after theory formation. For the induction of definitions, a unary-first search was used and we implemented the lookahead mechanism. For the scientific discovery application, a heuristic search, based on the novelty heuristic (see [5]) was employed and the ability to data mine the Encyclopedia was added. For the problem solving application, a different heuristic search was used and the ability to notice generalisations of the given concept was added. For the puzzle generation, an exhaustive breadth first search was employed and the abilities to choose examples for the puzzle and check for rival solutions were implemented. Hence, while theory formation can provide the initial information for an application and varying the search should improve performance, further processing is required to complete the task.

4.1 A Further Comparison of Progol and HR

While HR and Progol can form similar concepts, they differ in how they can be applied to each problem. When learning definitions for concepts, Progol generates possible answers, then builds new answers from the ones which achieve most compression first. On the other hand, HR does not use the given concept to choose which concepts to build on, until the answer has effectively been found and the lookahead mechanism enables it to take a shortcut to the answer. Without tweaking Progol, it appears that if there are few positive examples of a concept, Progol will not consider complicated definitions for them, as this achieves no compression. This may be a drawback for learning mathematical concepts, where the definitions are often fairly complicated, yet the examples scarce. In contrast, HR will carry on regardless of the complexity of concepts being formed, until an answer is found, or until it runs out of memory, etc. On the other hand, Inductive Logic Programming is a much more powerful technique than HR's lookahead mechanism, because this mechanism does not drive the search until the search is nearly over.

HR's application to scientific discovery was slightly different to Progol. Progol was used to find possibly complicated definitions for scientific concepts, where a categorisation into positive and negative examples was known beforehand. These definitions were, in some cases, interpreted as rules and used to explain the phenomena differentiating the positive and negative examples. Progol has had much success with this approach in many areas, in particular chemistry, biology and medicine. HR's approach to discovery was more exploratory, because we used it to identify concepts new to us. We did not supply HR with any information about the concepts we hoping it would find (such as a categorisation into positive and negatives) other than the fundamental concepts in the domain, e.g. divisors. Because there are so many concepts in a domain, HR had to identify which were interesting during its search so that it could use a heuristic search to reach more interesting concepts. More than this, after HR had found a concept which was missing from the Encyclopedia of Integer Sequences, it mined the Encyclopedia to find interesting conjectures to add further interest to the concept. In contrast, for Progol, there was no need to find reasons why the definitions were interesting, because the fact that one had been found at all to explain the observed phenomena was interesting in itself.

It is more difficult to comment on the problem solving and puzzle generation applications, because we have yet to study how Progol would be best applied to these problems. We have mainly suggested how Progol could be used in terms of applying its definition induction techniques to the problem at hand and we have not looked at any clustering ability Prolog may have, which may be a more suitable approach. Problem solving may be problematic for Progol, because it involves finding a definition not for the concept supplied, but for a generalisation of that concept. We have suggested a macro use of Progol, where negative examples are moved to the positives and a definition sought, which would produce a generalisation. However, this may turn out to be computationally expensive because there are many choices for the positives and negatives. Similarly with puzzle generation, we suggested that Progol try to learn definitions for given sets of examples and then show that no rival solution occurs.

For both the problem solving and puzzle generation applications, we have used HR to find a concept first, with the examples found afterwards. In contrast, our suggested use of Progol has been the opposite — to find the examples first, then find a definition which fits them. Until we perform more experiments with Progol, we cannot determine which approach is better for the two problems. However, in the case of puzzle generation, it is unlikely that a human writing a puzzle would start by writing down four examples, then try to find a concept embedded within them. However, we may find that a constraint based approach is more effective for puzzle generation.

5 Conclusions and Future Applications

Progol is generally used to induce a definition from a set of positive and negative examples, e.g. a definition for a subset of trains which are eastbound which distinguishes them from the westbound trains. This is a reactive process — a concept is immediately sought which defines the examples. It is possible to imagine another scenario, [3] whereby the program is given the same set of 10 trains and predicates describing them, but is allowed, say an hour, to prepare for an east-west question of the above nature. One effective way for a program to spend its time would be to invent many concepts related to trains, in particular, ways of classifying trains into a positive and a negative class. This is a more pro-active machine learning task, where the emphasis is on studying the trains rather than trying to learn a particular feature of them.

We gave the task to study trains as described above to HR, and in one hour it produced 160 specialisation concepts. There are only 638 ways to split 10 objects into two classes,[1] so if the user chose any subset of trains at random, there would be a one in four chance that HR could supply a reason why those trains were eastbound (and the others were not). We have performed similar pro-active learning tasks in number theory, using an agency of theory formation programs [6].

We have shown that theory formation can be applied to different learning tasks and highlighted the task involved, the additional functionality implemented in HR and the role of the user. While we make no claims that theory formation is the best way to approach these problems, we hope to have shown that it can be a useful tool for tasks involving learning. We have also compared HR to Progol both in terms of the concepts they form and their application (or proposed application) to the problems described. We have shown that Progol covers all the concepts that HR can form, but, even though HR was developed specifically in mathematical domains, only one of its production rules corresponds to additional background information in Progol. We have also suggested that for tasks such as puzzle generation, where it is necessary to find a set of concepts rather than just one and problem solving, where it is necessary to find a concept for which the

[1] See sequence A027306 in the Encyclopedia of Integer Sequences [22].

categorisation is not known, theory formation may be more applicable than the definition-inducing functionality that Progol mainly employs. However, we have not tested Progol in these areas and we do not comment in general on whether Progol could be employed to generate puzzles or solve problems of the type discussed above.

In future, we hope to apply HR to constraint satisfaction problems (CSPs), by automatically generating new constraints for a particular CSP. Each conjecture HR makes can, in principal, be turned into a new constraint for the CSP. However, certain conjectures will be less effective than others because they produce less propagation of constraints, and we will enable HR to decide which conjectures to add as constraints. We also hope to apply HR to automated theorem proving, whereby the user supplies a conjecture and requires a proof. We intend to test whether some initial theory formation before a proof attempt can decrease the time taken to prove a theorem. The theory produced would supply lemmas about the concepts in the conjecture statement which could be useful for the proof. As with CSPs, it will be necessary for HR to determine whether or not a lemma would be useful for a particular theorem. By applying HR to different problems, we hope to show that exploratory theory formation of the type HR undertakes embodies an important intelligent activity which has many uses in Artificial Intelligence.

Acknowledgments

I would like to thank Alan Bundy and Toby Walsh for continued detailed input to the HR project. I would also like to thank Stephen Muggleton, Chris Bryant and Richard Greaves for their in-depth discussions about Progol, HR and chemistry. Thanks also to Herbert Simon for enthusiastic discussions about the prospects for automated puzzle generation and problem solving.

References

[1] G Caporossi and P Hansen. Finding relations in polynomial time. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, 1999.

[2] S Colton. Refactorable numbers - a machine invention. *Journal of Integer Sequences*, 2, 1999.

[3] S Colton. Assessing exploratory theory formation programs. In *Proceedings of the AAAI-2000 workshop on new research directions in machine learning*, 2000.

[4] S Colton. Automated plugging and chugging. In M Kerber and M Kohlhase, editors, *Proceedings of the Eighth Symposium on the Integration of Symbolic Computation and Mechanized Reasoning*, 2000.

[5] S Colton, A Bundy, and T Walsh. HR: Automatic concept formation in pure mathematics. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, 1999.

[6] S Colton, A Bundy, and T Walsh. Agent based cooperative theory formation in pure mathematics. In *Proceedings of the AISB-00 Symposium on Creative & Cultural Aspects and Applications of AI & Cognitive Science*, 2000.

[7] S Colton, A Bundy, and T Walsh. Automatic identification of mathematical concepts. In *Machine Learning: Proceedings of the 17th International Conference*, 2000.

[8] S Colton, A Bundy, and T Walsh. Automatic invention of integer sequences. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, 2000.

[9] S Epstein. On the discovery of mathematical theorems. In *Proceedings of the International Joint Conference on Artificial Intellignce*, 1987.

[10] S Fajtlowicz. On conjectures of Graffiti. *Discrete Mathematics 72*, 23:113–118, 1988.

[11] R King, S Muggleton, and M Sternberg. Drug design by machine learning: The use of inductive logic programming to model the structure-activity relationships of trimethoprim analogues binding to dihydrofolate reductase. *Proc. of the National Academy of Sciences*, 89(23):11322–11326.

[12] D Lenat. AM: Discovery in mathematics as heuristic search. In D Lenat and R Davis, editors, *Knowledge-Based Systems in Artificial Intelligence.* McGraw-Hill Advanced Computer Science Series, 1982.

[13] W McCune. The OTTER user's guide. Technical Report ANL/90/9, Argonne National Laboratories, 1990.

[14] W McCune. A Davis-Putnam program and its application to finite first-order model search. Technical Report ANL/MCS-TM-194, Argonne National Laboratories, 1994.

[15] R Michalski and J Larson. Inductive inference of VL decision rules. In *Proceedings of the Workshop in Pattern-Directed Inference Systems (Published in SIGART Newsletter ACM, No. 63)*, 1977.

[16] S Muggleton. Inductive Logic Programming. *New Generation Computing*, 8(4):295–318, 1991.

[17] S Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.

[18] S Muggleton and L De Raedt. Inductive Logic Programming: Theory and methods. *Logic Programming*, 19-20(2):629–679, 1994.

[19] S Muggleton and D Page. A learnability model for universal representations. *JACM*, submitted 1999.

[20] J Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[21] H Simon and A Newell. Heuristic problem solving: The next advance in operations research. *Operations Research*, 6(1), 1958.

[22] N Sloane. The Online Encyclopedia of Integer Sequences. *http://www.research.att.com/~njas /sequences*, 2000.

[23] R Valdés-Pérez. Machine discovery in chemistry: New results. *Artificial Intelligence*, 74:191–201, 1995.

[24] S Wolfram. *The Mathematica Book, Fourth Edition.* Wolfram Media/Cambridge University Press, 1999.

[25] P Zeitz. *The Art and Craft of Problem Solving.* John Wiley and Sons, 1999.