On The Notion Of Interestingness In Automated Mathematical Discovery

Simon Colton and Alan Bundy
Institute of Representation and Reasoning
Division of Informatics, University of Edinburgh
80 South Bridge, Edinburgh. EH1 1HN.
simonco@dai.ed.ac.uk, bundy@dai.ed.ac.uk

Abstract

We survey five automated discovery programs working in mathematics, by looking in detail at the discovery processes they illustrate, summarising the successes they've had and by focusing on how they estimate the interestingness of concepts and conjectures. We then extract some common notions about the interestingness of conjectures and concepts. We detail how empirical evidence is used to give plausibility to conjectures, and the different ways in which a concept or conjecture can be thought of as novel. We also detail how programs assess how surprising and complex a conjecture statement is, and the different ways in which the applicability of a concept or conjecture is used. Finally we note how a user can set tasks for the program to achieve and how this affects the calculation of interestingness.

1 Introduction

There has been some recent progress in surveying and extracting general principles of machine discovery in science, for example Langley (1998), Valdés-Pérez (1999). We aim to add to this by surveying five programs developed to perform discovery in mathematics. We restrict our discussion to programs whose main objective is to invent concept definitions and make conjectures in pure mathematics. This leaves out automated theorem provers (which discover proofs), and programs which discover mathematical results in other domains, such as the very important BACON programs, Langley et al. (1987). To compare and contrast the discovery programs, we detail what the project aims were, how the program worked and what contributions the programs made to mathematics and the understanding of mathematical discovery. We pay particular attention to the measures employed to estimate how interesting a concept or conjecture is.

Deciding whether something is interesting or not is of central importance in automated mathematical discovery, as it helps determine both the search space and search strategy for finding and evaluating concepts and conjectures. Best-first searches using assessments of interestingness are often needed to effectively traverse large search spaces. When it becomes clearer what results are interesting, instead of just ignoring or discarding dull concepts and conjectures, the search space can be tailored to avoid some of them completely. Estimating interestingness is difficult because often it has to be done immediately after a concept or conjecture has been introduced, whereas the true interestingness of results and definitions in mathematics may only come to light much later. Therefore, a heuristic search has to be careful not to throw away anything which may turn out to be useful later on.

In §3, we identify six reasons why a concept or conjecture might be considered interesting. We detail how the programs use empirical evidence to cut down on the number of false conjectures made. We show how the novelty of a conjecture can be determined by whether it, or an isomorphic conjecture, has been seen before, or whether it follows as an obvious corollary to a previous conjecture, and we detail the different ways in which a concept can be thought of as novel. We note that being surprising is a desirable property of conjectures and concepts and we show how programs will avoid making conjectures which are just instances of tautologies, and how they can assess the surprisingness of a conjecture or concept. We define the applicability of concepts and conjectures to be the subset of models to which they bear some relevance, and show that this measure can be used in a variety of ways. We also detail how programs can assess complexity and tailor their search strategies to find the least complex concepts and conjectures first. Finally, we look at how a user can set a program a particular task to achieve and how interestingness can be measured with respect to that task.

By looking in detail at five discovery programs and extracting some common ways by which the interestingness of concepts and conjectures is assessed, we will be able to suggest possible ways for future programs to measure interestingness. We will also note that the novelty, intelligibility and plausibility measures set out in Valdés-Pérez (1999) by which humans can assess the output of discovery programs are similar to those by which the programs assess the interestingness of their results internally.

2 Machine Discovery Programs

The five programs we discuss in detail are the AM program which worked in elementary set and number theory, the GT program which worked in graph theory, the Graffiti program which is used in graph theory, the plane geometry system from Bagai et al, and the HR program which works mainly with finite algebras. For clarity, no other programs are discussed.

2.1 The AM Program

The AM program, written by Douglas Lenat, performed concept formation and conjecture making in elementary set theory and elementary number theory, as described in Lenat (1976) and Davis and Lenat (1982). Starting with 115 elementary concepts such as sets and bags, AM would re-invent set theory concepts like subsets and disjoint sets, and number theory concepts such as prime numbers and highly composite numbers (with more divisors than any smaller integer). AM would also spot some well known conjectures, such as the fundamental theorem of arithmetic and Goldbach's conjecture - that every even number is the sum of two primes.

Concepts were given a frame representation with 25 facets to each frame, and none, one or multiple entries for each facet. Some of the facets were: (i) a definition for the concept (ii) an algorithm for the concept (iii) examples of the concept (iv) which other concepts it was a generalisation/specialisation of, and (v) conjectures involving the concept. AM repeatedly performed the task at the top of an agenda ordered in terms of the interestingness of the tasks. Each task involved performing an action on a facet of a concept. Usually the action was to fill in the facet, for example, find some other concepts which are specialisations of the concept or find some conjectures about the concept, but the action could also be to check the facet, eg. check that a conjecture was empirically true.

To perform a task, AM would look through its database of 242 heuristics, choose those which were appropriate to the task and perform each of the sub-tasks suggested by the chosen heuristics. Some sub-tasks detailed how to perform the overall task at hand, but they were not limited to that. Some sub-tasks would put new tasks on the agenda (which was how the agenda was increased). Some of the new tasks were to invent new concepts and when these were added to the agenda, AM would immediately create the frame for the new concept. This was because knowledge present at the time of suggesting the new concept was needed to fill in some of the facets of the concept. AM only filled in information at this stage which took little computation, such as a definition and examples, and a task was put on the agenda to fill in each of the other facets of the newly formed concept.

Among the new concepts AM would suggest were: (i) specialisations, eg. a new function which was a previous one specialised to have equal inputs, (ii) generalisations (iii) extracted from the domain/range of a function, eg. those integers output by a function (iv) inverses of functions (v) compositions of two functions. Some tasks on the agenda were to find conjectures about a concept, including finding that (a) one concept was a specialisation of another (b) the domain/range of a concept was limited to a particular type of object or (c) no integers of a particular type existed.

Because there could be as many as 4000 tasks on the agenda at any one time, AM spent a lot of its time deciding which it should do first. Whenever a heuristic added a task to the agenda, it would supply reasons accompanied by numerical values why the action, concept or facet of the task was interesting. AM then employed a formula involving the number of reasons and a weighted sum of the numerical values to calculate an overall worth for the task. The weighted sum gave more emphasis to the reasons why the concept was interesting than the reasons why the facet or action were interesting. When a heuristic was working out how interesting a concept was, it would collate and use another set of heuristics for the task. The heuristics which could measure the interestingness of any concept were recorded as heuristics 9 to 20 in Davis and Lenat (1982), and included:

[9] A concept is interesting if there are some interesting conjectures about it.

[13] A concept is dull if, after several attempts, only a couple of examples have been found.

[15] A concept is interesting if all the examples satisfy the rarely-satisfied predicate P.

[20] A concept is more interesting if it has been derived in more than one way.

(Note that these have been paraphrased from Lenat's originals). AM also had ways to assess the interestingness of concepts formed in a particular way, for example the interestingness of concepts formed by composing two previous concepts could be measured by heuristics 179 to 189, one of which was:

[180] A composition $F = GoH$ is interesting if F has an interesting property not possessed by either G or H.

AM would also measure the interestingness of conjectures, so that it could correctly assess tasks relating to the conjectures facets of concepts. Heuristics 65 to 68 seem to be the only heuristics which do this, for example:

[66] Non-existence conjectures are interesting.

At any stage during a session, the user could interrupt AM and tell it that a particular concept was interesting. Lenat says in Davis and Lenat (1982) that users could "kick AM in one direction or another", and "the very best examples of AM in action were brought to full fruition only by a human developer". Many of AM's heuristics were designed to keep the focus on such chosen concepts, by spreading around the interest the user had shown in them. For example, these heuristics keep the attention on concepts and conjectures related to interesting concepts:

[16] A concept is interesting if it is closely related to a very interesting concept.

[65] A conjecture about concept X is interesting if X is very interesting.

In fact, AM could make a little interestingness go a long way: of the 43 heuristics designed to assess the interestingness of a concept, 33 of them involve passing on interestingness derived elsewhere.

There has been much debate about the AM program. Ritchie and Hanna (1984) were particularly critical of the methods AM used and the accuracy of Lenat's description of AM. The main contribution of Lenat's work is an inspiration for how computers could do mathematics, ie. by creating concepts and conjectures of many different types and using heuristic methods such as analogy and symmetry to explore a domain.

2.2 The GT Program

Th GT program by Susan Epstein performed concept formation, conjecture making and theorem proving in graph theory, as described in Epstein (1987) and more fully in Epstein (1988). Given just the concept of a graph, GT would re-invent graph properties, such as being acyclic, connected, a star or a tree, (as shown in figure 1 below).

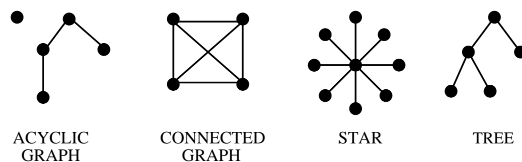

Figure 1: graph properties re-invented by GT

Also, given a set of user-defined concepts describing graph properties, GT would make conjectures such as:

- A graph is a tree iff it is acyclic and connected.

GT successfully illustrated a possible mechanism for automated discovery in mathematics involving both deductive and inductive reasoning. This was possible because GT represented graph properties in a carefully thought out way developed in Epstein's PhD thesis, Epstein (1983), which allowed example generation, theorem proving and concept formation.

Each graph property was represented as a triple, $< f, S, \sigma >$, consisting of a set of base cases, S, a constructor, f, and a set of constraints for the constructor, σ, which together detailed the recursive construction of graphs from the base cases. For example, to define the star property above, the base cases would be just the trivial graph (with one vertex, no edges) and the constructor would add one vertex and an edge between the new vertex and an old vertex, subject to the single constraint that the old vertex must be on more edges than any other vertex. Epstein was able to prove that 42 classically interesting graph theory concepts, including cycles, Eulerian graphs and k-coloured graphs, could be represented in this manner in a sound and complete way.

This representation could be used to generate examples of a concept (Epstein called this 'doodling') by starting with the base cases and repeatedly applying the constructor, subject to the constraints. Deduction was possible by proving that one graph property subsumed another [see Epstein (1988)], or by showing that no graphs could have two particular properties. Concept formation was possible by: (a) *specialising* a previous concept by removing base cases, restricting the constructor, or strengthening the constraints, (b) *generalising* a previous concept by adding base cases, expanding the constructor, or by relaxing the constraints, or (c) *merging* properties A and B, for example creating a new graph property with A's base cases and constructor, but the constraints of both A and B, [subject to some conditions].

GT worked by repeatedly completing one of six types of project: (i) generate examples of graphs with certain properties, (ii) see if one property subsumed another (iii) see if two properties were equivalent, (iv) see if a merger between two properties would fail, (v) generalise a concept and (vi) specialise a concept. Each project was placed on an agenda following various rules:

- If a property has few examples in the database, then immediately generate more examples for it by 'doodling'.
- Two properties, P and Q, are better candidates for projects (ii) or (iii) above if the set of base cases for P and Q are similar. Two sets are most similar if they are equal, less similar if one is a subset of the other and less similar still if they only have a non-trivial intersection.
- Only perform specialisation or generalisation projects with a concept before conjecture-making projects if the user has flagged the concept as a 'focus' (see later).

As an overview, if a conjecture project was at the top of the agenda, before trying to prove the conjecture, GT would first see if there was empirical evidence against the conjecture, using the generated examples of the graphs [note that a conjecture was suggested only using the base cases]. If the project was to check a merger conjecture, then the merge step would take place, and only if no graphs of the merged type could be produced would an attempt be made to prove the conjecture. If a generalisation or specialisation project was at the top of the agenda, it would be carried out and some effort expended to generate examples of the new concept.

Focus concepts could be specified by users if they were particularly interested in them, and, as well as restricting concept formation only to the focus concepts, GT would only make conjectures involving the focus concepts. GT rated certain newly formed concepts as uninteresting and discarded them. For example, if a concept was a generalisation to a focus concept, but no example graphs could be produced which were not examples of the focus concept, the new concept was discarded. Also, if only a few graphs could be generated with a newly formed

property, the new concept was discarded. By identifying the routine of ordering which conjectures to look at first, attempting to make and prove the conjectures, and performing concept formation only with the most interesting concepts, Epstein's model of discovery was successfully implemented and produced theories containing different kinds of conjecture and their proofs and concepts and graphs not present at the start of the session.

2.3 The Graffiti Program

The Graffiti program, by Siemion Fajtlowicz, makes conjectures of a numerical nature, mainly in graph theory, as described in Fajtlowicz (1988), and more recently in Larson (1999). Given a set of well known, interesting graph theory invariants, such as the diameter, independence number, rank or chromatic number, Graffiti uses a database of graphs to empirically check whether one sum of invariants is less than another sum of invariants. If a conjecture passes the empirical test and Fajtlowicz cannot prove it easily, it is recorded in the "writing on the wall", some of which is publicly available, Fajtlowicz (1999), and Fajtlowicz forwards it to interested graph theorists. These types of conjecture are of substantial interest to graph theorists because (a) they often provide a significant challenge to resolve and (b) calculating invariants is often computationally expensive, so any bounds on their values are useful. As an example, the 18th conjecture in the writing on the wall states that, for any graph, G,

$$\text{chromatic_number}(G) + \text{radius}(G) \leq \text{max_degree}(G) + \text{frequency_of_max_degree}(G)$$

The empirical check is time consuming, so Graffiti employs two techniques, called the *beagle* and *dalmation* heuristics, to discard certain trivial or weak conjectures before the empirical test:

The *beagle* heuristic discards many trivially obvious theorems, including those of the form $\mathrm{i}(G) \leq \mathrm{i}(G) + 1$. Note that invariants which are a previous invariant with the addition of a constant are used to make stronger conjectures. The beagle heuristic uses a semantic tree of concepts to measure how close the left hand and right hand terms are in a conjecture, and rejects those where the sides are semantically very similar.

The *dalmation* heuristic checks that a conjecture says something more than those made by Graffiti previously. To use the dalmation test for a conjecture of the form $p(G) \leq q(G)$, Graffiti first collates all conjectures it has ever made of the form $p(G) \leq r_i(G)$. Then, to pass the dalmation test, there must be a graph, G_0, in Graffiti's database which for all the r_i, $q(G_0) \leq r_i(G_0)$. This means that, for at least one graph, $q(G)$ gives a stronger bound for $p(G)$ than any invariant suggested by a previous conjecture, so the present conjecture does indeed say something new about Graffiti's graphs.

Another efficiency improving technique employed by Graffiti is to restrict the database of graphs to only those which have at one stage been identified as a counterexample to one of Graffiti's conjectures. A third efficiency technique is to remove by hand any previous conjectures which are subsumed by a new conjecture. For example, Fajtlowicz would move the old conjecture $i(G) \leq j(G) + k(G)$ to a secondary database, if the conjecture $i(G) \leq j(G)$ was made. However, if the latter conjecture was subsequently disproved, the former conjecture would be restored.

As Fajtlowicz adds concepts to Graffiti's database, the writing on the wall reflects the new input, eg. conjectures 73 to 90 involve the coordinates of a graph. Fajtlowicz can also direct Graffiti's search by specifying a particular type of graph he is interested in. For example, conjectures 43 to 62 are about regular graphs. To enable this kind of direction, Fajtlowicz informs Graffiti of the classification of its graphs, into, say, regular and non-regular graphs. Then, if Graffiti bases its conjectures on only the empirical evidence supplied by the regular graphs, the conjectures will only be about those graphs. To stop Graffiti re-making all of its previous conjectures, the *echo* heuristic uses semantic information about which graph types are a subset of which others, and rejects conjectures about the chosen type of graph if there is a superset of graphs for which the conjecture is also true.

In terms of adding to mathematical knowledge, the Graffiti program has been extremely successful. Its conjectures have attracted the attention of scores of mathematicians, including many luminaries from the world of graph theory. There are over 60 graph theory papers which investigate Graffiti's conjectures. While Graffiti owes some of its success to the fact that the inequality conjectures it makes are of a difficult and important type, this should not detract from the simplicity and applicability of the methods and heuristics it uses.

2.4 Bagai et al's System

The program developed by Rajiv Bagai et al, described in Bagai et al. (1993), worked in plane geometry and constructed idealised diagrams and proved conjectures that certain diagrams could not be drawn. Each concept was a situation in plane geometry involving points and lines and relations between the points and lines, such as a point being on a line or two lines being parallel. For example, a parallelogram and its diagonals, as in figure 2 below [taken from Bagai et al. (1993)], could be described by stating that there were four ingredient points, A, B, C and D, six lines (one between each pair of distinct points) and two relations, namely that lines AB and CD were parallel and that lines AC and BD were parallel

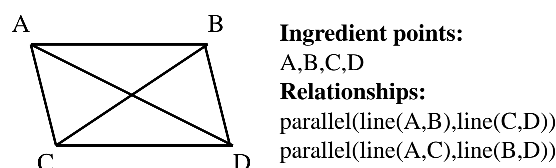

Figure 2: a parallelogram and diagonals, and its representation

Starting with an empty situation, constructions like the parallelogram were made by adding new ingredient points and new relations to a previous situation. Each time a new relation was added, a conjecture was made that the resulting situation was inconsistent, ie. that it was not possible to draw an example of the situation. To prove the conjecture, the situation was turned into a collection of polynomials and inequalities which were passed to an efficient theorem prover, Chou (1984). If the theorem prover could not find complex solutions to the polynomial then the situation was indeed inconsistent. However, if the theorem prover did find complex solutions, this said nothing about the consistency of the situation, and the conjecture was discarded to preserve the soundness of the theories produced.

Many methods were employed to reduce the number of times the system used the theorem prover. Firstly, only consistent situations were built upon, as a situation which was an extension of an inconsistent situation would itself be inconsistent. By also restricting to only adding one relation at a time, if the situation produced was inconsistent, the additional relation must have caused the inconsistency. This enabled better presentation of the theorems, eg. if the condition that lines AD and BC were parallel was added to the parallelogram situation above, this would cause an inconsistent situation. As the inconsistency was caused by the new relation, instead of just stating that a parallelogram with parallel diagonals cannot be drawn, the system could say that:

Given a parallelogram, the diagonals cannot be parallel.

Another way to reduce the time spent using the theorem prover was to avoid proving the inconsistency of a situation which was isomorphic to a previous one. Two situations were isomorphic if a permutation of the ingredient points of the first produced the second. To get around this problem, whenever a situation was built, all of its isomorphic situations were also built, so that they could be recognised if re-constructed by a different route later on. Also, to cut down on the occurrences of later theorems which implied earlier theorems a breadth first search was used where a step could only be the addition of either a single ingredient point or a single new relation. This meant that the most general situations were constructed before the more specific ones and therefore the most general versions of theorems were produced first. Not only could the program re-discover well known results such as Euclid's 5th postulate, it also provides a very clear and concise theory for the automatic production of a subset of plane geometry concepts and a set of theorems about the non-existence of models for certain concepts.

2.5 The HR Program

The HR program by Colton et al, as described in Bundy et al. (1998) and more recently in Colton and Bundy (1999), was originally developed to perform concept formation in group theory. Starting with just a few definitions, HR can re-invent classically interesting concepts such as centres of groups, Abelian and Cyclic groups and orders of elements. HR works directly with the models of concepts (stored as data-tables), and constructs new concepts by taking the data-tables of old concepts and manipulating them using one of ten production rules to produce a new data-table. From information about how a concept was constructed, HR can generate a definition for the concept whenever one is needed.

HR encounters a combinatorial explosion because a single concept can often be transformed into around 20 new ones, and any pair of concepts can be combined into a third. A heuristic search is used which chooses the best concept to use in each concept formation step. HR has a variety of ways to measure concepts and a weighted sum of measures is taken to indicate an overall level of interestingness for the concept. The weights are set by the user and depend on what types of concepts they are looking for. One way to use HR is to supply a 'gold standard' categorisation of the groups in the database, and ask HR to find a function, the output of which will categorise the groups correctly (groups with the same output are put in the same category). HR can then measure how close each concept gets to this categorisation, by evaluating the proportion of pairs of groups that a concept correctly categorises. This approach can be effective, for example, given the isomorphic classification of the groups up to order six, HR found this function which correctly classifies them: $f(G) = |\{(a, b, c) \in G^3 : a * b = c \ \& \ b * c = a\}|$.

If the user has no particular task in mind, they can ask HR to explore the domain. HR has certain measures which indicate desirable properties of a concept, and users can stress some of these if they wish. The *parsimony* measure of a concept is inversely proportional to the size of the data-table for the concept. The data in a table corresponding to a particular group can be used to describe that group, and so a small table is advantageous as this means more parsimonious descriptions. HR can also assess the *novelty* of a concept, which is inversely proportional to the number of times the categorisation produced by a concept has been seen already, (with more unusual categorisations being more interesting). Finally, HR can measure the *complexity* of a concept which is inversely proportional to the number of old concepts appearing in its construction path. This gives a rough indication of how complicated the definition of a concept will be, and more concise definitions are desirable.

HR can make conjectures by spotting that the data-table of a newly formed concept is exactly the same as a previous concept, and conjecturing that the concepts are equivalent. When this happens, definitions for each concept are generated and used to write the conjecture in a way acceptable to the OTTER theorem prover, McCune (1990), which HR asks to prove the conjecture. For example, when HR invents the concept of elements, a, for which $a*a = a$, it spots that the new data-table is the same

as the one it has for the user-given concept of the identity element, and the following conjecture is generated:

$$\forall a, (a = id \iff a * a = a).$$

This is broken into $\forall a, (a = id \longrightarrow a * a = a)$ and $\forall a, (a * a = a \longrightarrow a = id)$, which are both passed to and easily proved by the theorem prover, OTTER. Before passing a conjecture to OTTER, HR uses some simple deductive techniques to check whether the conjecture follows easily from those already proved.

HR has a set of 'sleeping concepts', such as the trivial group, and when a concept is conjectured to be the same as these, the conjecture is flagged so that the user can pay special attention to it (or choose to ignore it). If the interestingness of conjectures and proofs can be estimated, then the average interestingness of the theorems a concept appears in can be taken as a new measure for the interestingness of the concept itself. Conjectures are assessed in two ways. Firstly, the *surprisingness* of a conjecture measures how different the two (possibly) equivalent concepts are, by evaluating the proportion of concepts which appear in the construction path of one but not both of the concepts. This gives some indication of how different looking the definitions of the equivalent concepts are going to be. Secondly, if a conjecture is proved, OTTER will provide a *proof length* measure in its output, which gives some indication of the difficulty of the proof.

If the equivalence of two definitions is proved, HR uses this fact to re-assess the concepts involved, and keeps only the least complex definition for the concept. If OTTER cannot prove a conjecture, HR passes it to the MACE model generator, McCune (1994), which is asked to find a single counterexample to the conjecture. If MACE is successful, the counterexample is added to HR's database and all previous concepts and measures are re-calculated, giving HR a better idea of the theory it is exploring. All future conjectures will be based on the additional data provided by the new group. HR's theory formation is general enough to apply to any finite algebra and HR can bootstrap the process, that is, it can start with just the axioms of the algebra, and end with a theory containing models, definitions, open conjectures, theorems and proofs. The concept formation is general enough to apply to different domains, including graph and number theory.

Indeed, HR's biggest success so far has come in number theory, where it invented the concept of refactorable numbers, the first examples of which are:

$$1, 2, 8, 9, 12, 18, 24, 36, 40, 56, 60, 72, 80, 84, 88, 96, \ldots$$

These are defined as those integers where the number of divisors is itself a divisor. This concept was novel because it was missing from and subsequently added to the online-encyclopedia of integer sequences, Sloane (1998), which contains over 45,000 integer sequences. Refactorables are also interesting because there are many provable properties about them, for example, all odd refactorables are square numbers. Results about refactorables have been published in the mathematical literature, Colton (1999).

3 Assessing the Interestingness of Conjectures and Concepts

We cannot discuss measures of interestingness without addressing how the measures are used. For example, one program might say that the concept of even primes is interesting because a conjecture can be made that 2 is the only one, whereas another program might say that they are dull because only one example of them is known. Here the same measure has been used (see §3.4 below), but different conclusions have been drawn. Therefore we have to clearly separate the measures for interestingness from their uses. One common use of interestingness is to improve the efficiency of the programs. To save time checking and proving conjectures, certain conjectures are discarded before even checking them empirically, and the reason to perform an empirical check is, of course, to cut down on the time spent trying to prove false conjectures.

Another common use of interestingness is to improve the appeal of the output for the user. It is not possible to avoid all uninteresting concepts or conjectures when constructing a theory and interestingness measures can be used to filter the output depending on the user's needs. Also, measures of interestingness can guide the search so that the program can make informed progress into the space and find interesting concepts that it might take a longer time to find with an exhaustive search. A more specific use of interestingness measures is to predict in advance how difficult a conjecture is going to be to prove, which, in all but some trivial circumstances is not easy to do. Finally, interestingness measures can be used to steer the concept formation towards a particular concept which performs a user-defined task. Having identified some uses for interestingness, we can detail certain general measures and look at how each one is used.

3.1 Empirical Plausibility of Conjectures

A conjecture is likely to be uninteresting if the empirical evidence a program has provides counterexamples. This does not mean that false conjectures in general are uninteresting, as the production of counterexamples is a worthwhile pursuit. However, if a counterexample is found in the data a program has, the conjecture cannot provide this worthwhile pursuit. Only the system developed by Bagai et al makes conjectures which have not been first verified by some empirical evidence. In this case, the efficiency and power of the theorem prover and the nature of the idealised geometrical domain make it unproductive to look for counterexamples. The AM program is the only one which doesn't immediately discard a conjecture proved false by empirical evidence. In this case, an attempt is made to alter the conjecture to make it fit the data. One way to do this is to exclude what AM calls 'boundary' integers, so for example, the conjecture that 'all primes are odd' becomes the conjecture that 'all primes except 2 are odd'.

HR and Graffiti use all of their data at once. HR uses its data to spot (ie. suggest) a conjecture, so by the time a conjecture has been made, the empirical check has been completely performed. Similarly, once a conjecture has been suggested by other means to Graffiti, all the empirical evidence is used to discard it. Note however, that both of these programs keep the amount of empirical data down to a minimum because they only store models which have been generated as counterexamples to previous conjectures. GT employs a more efficient system because a conjecture is suggested by the small amount of empirical evidence in the set of base cases, and only those conjectures passing this test are checked against all the examples for the concepts. Similarly, AM will make a conjecture based on a little empirical evidence, then try to generate more models to disprove the conjecture.

3.2 Novelty

Because of the redundancy often inherent in searches for concepts and conjectures, it is important to be able to spot when a repetition has occurred, and each program either tailors its search to reduce repetitions, or can spot them when they occur. Here, the programs are measuring the novelty of a concept or conjecture statement, and rejecting those which have been seen already. The Graffiti program goes to the length of storing conjectures between sessions. Another issue of novelty in programs searching for conjectures is whether a theorem follows as an obvious corollary to a stronger theorem, in which case, the weaker result does not say anything particularly new. Graffiti works hard to show that a new conjecture says something more than all the previous ones, by checking that there is at least one graph for which the inequality in the conjecture is stronger than all the previous ones (the dalmation heuristic), and by checking that the conjecture is not implied by previous ones (the echo heuristic). Spotting the implication of one conjecture by another is also used in Graffiti to improve efficiency: if a later conjecture turns out to be stronger than a previous one, the earlier one is removed, hence saving Graffiti time later when it looks through its old conjectures.

The HR program deals with the implication of one conjecture by previous ones by extracting and attempting to prove all stronger conjectures than the one it is considering. For example, if interested in the conjecture P & $R \longrightarrow Q$, HR will first use some simple deductive techniques to see if it follows as a corollary to the results it has already proved. If not, it will ask OTTER to prove $P \longrightarrow Q$, and $R \longrightarrow Q$ and if either turns out to be true, the weaker, original conjecture is discarded and the stronger conjecture is kept. The most efficient way to deal with one conjecture implying another is to tailor the search to produce the most general conjectures first, so there is less chance that later conjectures will imply the earlier ones. This technique is used in the system from Bagai et al, but they point out in Bagai et al. (1993) that it is still possible to produce a later conjecture which implies an earlier one.

A concept can easily be shown to be novel with empirical evidence. For example, if a function produces some output for a given input that no other function produces, it must be novel. Given only a limited amount of data though, it is more difficult to tell that two concepts are indeed the same. Bagai et al's system (which has no data available at all) tackles this by generating all possible isomorphic concepts whenever a new concept is introduced, so that if an isomorphic concept to the new one is reached by another route, the system will spot this. HR's conjecture making abilities rely on the fact that a proof is often needed to tell that two concept definitions are in fact equivalent, and, like AM, HR assesses concepts as more interesting if they were derived in two or more different ways. If a concept isn't the same as one already in the theory, it is possible to assess how much it differs from the others, using certain properties of it. AM, for example, gave extra interestingness to newly formed concepts, ie. those with the special property of being recently invented. The HR program assesses the novelty of a concept in terms of the novelty of the categorisation of groups it gives. It is important to make the distinction between one concept having two properties (eg. two definitions), which is often interesting, and two concepts sharing the same property (eg. a categorisation), which often detracts from the interestingness of both.

3.3 Surprisingness

As portrayed in Fajtlowicz (1999), when asked what makes a good conjecture, the mathematician John Conway said without any hesitation: "it should be outrageous". This is good advice, and in some cases an assessment of how surprising a conjecture is can be automated. The least surprising conjectures are those which are just instances of tautologies. For example, given objects of any type, A, and predicates of any nature, p and q, the conjecture

$$\forall A,\ not(not(p(A))) \iff p(A)$$

is always going to be true, and conjecture finding programs should avoid making these and similar conjectures.

To avoid making tautologies of a particular type, GT did not make subsumption conjectures if one of the concepts was a specialisation of the other. The HR program avoids certain tautologies by forbidding certain series of concept formation steps, eg. not allowing two negation steps in succession. Graffiti uses a semantic tree to measure how different invariants i and j are in the conjecture: $\forall$ graphs G, $i(G) \leq j(G)$, and the beagle heuristic discards many tautology conjectures which involve very similar concepts, such as $i(G) \leq i(G) + 1$. Whereas Graffiti uses its measure for surprisingness only to discard conjectures, when the HR program makes a a conjecture that two concept definitions are equivalent, it has semantic information about those concepts, so can tell how different they are, giving an indication of how surprising the conjec-

ture is. HR uses the heuristic that concepts appearing in surprising conjectures are more interesting, which helps drive a best first search. While HR and Graffiti can estimate the surprisingness of conjectures, only AM measured the surprisingness of a concept: it gave extra interestingness to concepts which possessed an interesting property not possessed by its parents (see heuristic 180 in Davis and Lenat (1982), for example).

3.4 Applicability

The applicability of a predicate can be defined as the proportion of models in a program's database which satisfy the predicate. Also, the applicability of function can be defined as the proportion of models in a program's database which are in the domain of the function. This measure is somewhat analogous to the empirical plausibility of conjectures, but can itself be extended to cover conjectures: the applicability of a conjecture can be defined as the proportion of models in a program's database which satisfy the conjecture's preconditions. These measures are used in a variety of ways as follows.

In AM and GT, if a newly formed concept had low applicability (ie. few examples), a task was put on the agenda to generate some more models that it applied to. If a concerted effort to generate examples still resulted in a low applicability, GT would discard the concept as uninteresting, and AM would give it a low interestingness score. In the special case where the applicability was zero, (ie. no examples were found), both GT and AM would make the conjecture that none exist. The HR program makes similar non-existence conjectures, and other conjectures about the applicability of a concept, for example, that it is restricted to the trivial group. In Bagai et al's system, the whole point was to prove that certain concepts have no models (ie. situations are inconsistent), which is equivalent to showing that the applicability of a concept is zero. So we see that concepts with little or no applicability are often thought of as dull, but the conjecture that this is true is interesting.

Furthermore, in GT, if a generalisation step produced a concept with no greater applicability than the one it generalised, or if a specialisation step produced a new concept with a greater applicability than the one it was supposed to specialise, the new concept was discarded. In Graffiti, if the user has specified an interest in a particular set of graphs (ie. those with a particular quality), then if a conjecture is output which is applicable to a superset of that set, it is discarded as being too general (the echo heuristic). Also, in HR, the parsimony measure prefers concepts with small data-tables, and hence small applicability. This gives an emphasis to specialisation procedures and can be useful in controlling the search. Finally, the AM program used 'rarely satisfied predicates' - those with low applicability - to assess other concepts. For example, heuristic 15 from Davis and Lenat (1982) gave more interestingness to functions whose output always satisfied one of the rarely satisfied predicates that AM had come across. We see that applicability is a common measure, but how it is used is always determined by the context of the discovery task being attempted.

3.5 Comprehensibility and Complexity

As programs are intended to produce output understandable by the user, more comprehensible concepts and conjectures are usually more interesting. The GT and Bagai et al programs constructed concepts incrementally so that the most comprehensible ones were introduced first. The HR program employs the complexity measure which prefers concepts with smaller construction paths (which roughly relates to how long their definition will be). HR's best first search is not guaranteed to produce the least complex concepts first, so, if two concepts are proved to be equivalent, the least complex definition of the two is kept. Also in HR, concepts can be used to describe the groups in the theory, and the parsimony measure prefers concepts giving shorter, more concise descriptions.

The comprehensibility of a concept gives one, albeit shallow, indication of the complexity of that concept, and usually less complex, more comprehensible, concepts are desirable. An alternative way to assess the complexity of a concept is to evaluate how much information there is about it. AM counts how many conjectures there are involving a concept and uses this as a measure of the interestingness of the concept. The HR program goes one stage further and assesses not only the conjectures but also the proofs of them, and uses this to measure the interestingness of the concepts involved in the conjectures.

The novelty and surprisingness measures often prune easy to prove conjectures, because tautologies, conjectures which follow as an obvious corollary to previous theorems and those which are unsurprising have a greater likelihood of being easy to prove. Removing these conjectures will possibly increase the average difficulty to prove the conjectures remaining. However, this is separate from the issue of how difficult a conjecture is to understand. Preferring conjectures about less complex concepts will increase the overall comprehensibility of the theory, and choosing a simple format for conjectures can also help. For example, as noted in Valdés-Pérez (1999), it is easy to understand what Graffiti's inequality conjectures are saying. Further, the program from Bagai et al presents its theorems not as unsatisfiability results, but as relations which cannot occur once a situation has been set up, which is a more understandable format.

3.6 Achieving Particular Tasks

There are ways by which the user can explicitly express interest in particular concepts or conjectures. This is clear in the AM program which was designed as an interactive program where the user can interrupt the session at any time and express an interest in a particular concept. AM's

heuristics were set up to pay particular attention to this concept, and the limited number of concepts AM could produce in a session were heavily biased by the user's choice. Here we see that the user wants AM's concepts and conjectures to perform a particular task, namely to discuss the concept chosen by the user. This is taken a stage further in the GT and Graffiti programs, where the user can specify a 'focus' concept, and only conjectures involving the chosen concept will be produced. In GT's case, this also meant that specialisations or generalisations of only the focus concept would be attempted.

In the Graffiti program, all the proved conjectures give a bound for an invariant (which may save computation time), so we see that the search space has been designed with a task in mind. More explicit tasks are set for other programs. By giving a 'gold standard' classification of groups to HR, users are expressing an interest in concepts which achieve that categorisation. By giving concepts and conjectures particular tasks to achieve, a program can measure how close each comes to completing the tasks and use this to estimate interestingness, which will hopefully drive the best-first search towards something which achieves the task.

4 Conclusions

Assessing the interestingness of a concept or conjecture automatically is difficult because a program has to try to predict how much useful mathematics will result from an investigation of the concept or the attempts to prove the conjecture. Fermat's last theorem, for example, could easily have been relegated to the appendix of a number theory text if it had not been so difficult to prove. Also, as in discovery of any kind, it is often necessary to have expert knowledge to decide whether an invention has any far reaching implications or applications, as pointed out in Langley (1998).

By comparing and contrasting five machine discovery programs, all of which are forced to guide their search towards more interesting output and make instant decisions about the possible interestingness of conjectures and concepts, we have extracted some possible ways a program can measure the interestingness of the concepts and conjectures it makes. The empirical plausibility of conjectures, novelty and comprehensibility measures are analogous to qualities described in Valdés-Pérez (1999) to assess the output from machine discovery programs. Therefore, we see that automated attempts to estimate interestingness often check the internal plausibility, novelty and intelligibility of a theory in the ways a human might estimate the quality of a theory externally.

4.1 The Interestingness of Conjectures

In summary, it is often a good idea to prescribe a definite task for a concept or conjecture to achieve, and design measures of interestingness around this. If this is not possible, and an estimate of the interestingness of a conjecture is going to be made, some of the following points could be taken into consideration:

- The conjecture should be empirically true.

This can be achieved by only making conjectures backed up by all available data, suggesting a conjecture by some other means and then using all the data to discard the conjecture if necessary, or altering a conjecture so that any data which disproves it is no longer applicable.

- The conjecture should be novel w.r.t. previous ones.

This can be achieved by understanding and checking how two conjectures can be equal or isomorphic in the domain of interest. Also, conjectures which are implied by previous, stronger conjectures, should either be avoided by tailoring the search space, or rejected when found.

- The conjecture should be surprising in some way.

This can be achieved by avoiding or discarding well known tautologies, and by using information about the concepts discussed in the conjecture to estimate how unlikely the suggested relation between them is.

- The conjecture should discuss some non-trivial models.

This can be achieved by discarding any conjectures where the only models satisfying the preconditions are a trivial, or uninteresting set. Note that in certain circumstances, this kind of highly specialised conjecture may actually be of interest to the user.

- The conjecture should be understandable, but non-trivial to prove.

This can be achieved by assessing the number and diversity of concepts involved in a conjecture and removing overly complicated conjectures, or by fixing the search strategy to output the simplest conjectures first. Making conjectures in a well known format will help understandability. If the conjecture has been proved, the proof could be used to estimate how difficult the theorem was.

4.2 The Interestingness of Concepts

If an estimate of the interestingness of a concept is going to be made, some of the following points could be taken into consideration:

- The concept should have models.

This can be achieved by checking available data for models which satisfy a predicate or are in the domain of a function. If no models exist, an effort should be made to generate some. It may be necessary to prove that no models exist, and discard the concept (but keep the theorem).

- The concept should be novel w.r.t. previous ones.

To achieve this, the search should ensure that no two obviously isomorphic definitions can be made, and avoid paths which will ultimately lead to the same concepts.

Then, if the models of one concept are the same as another, the concepts are possibly equivalent, which should lead to a proof of this fact. If a concept is indeed semantically different to all the others, then the novelty of various properties (such as the way it categorises models) could be assessed.

- There should be some possibly true conjectures made about the concept.

By the qualification of possibly true conjectures, we note that while false conjectures about the nature of a concept are usually of no interest, an unsettled conjecture can often be more interesting than a proved theorem.

- The concept should be understandable.

This can be achieved by designing the search to construct concepts with the simplest definitions first, and by keeping the simplest definition when it has been proved that two concepts are equivalent.

- The concept should have a surprising property.

A surprising property may be something that isn't true of the parent concepts.

Building on the techniques for automated discovery that have been developed in artificial intelligence and cognitive science, and learning from the results of programs developed in mathematics, an effort can be made to write more programs which act as collaborators with working mathematicians. The production of intelligently suggested conjectures and concepts plays an integral and important part in developing a mathematical theory, and automating these processes is a worthy area for research. How programs estimate the interestingness of the concepts and conjectures they produce is central to building intelligent discovery programs, and we hope that the notions of interestingness derived here will be of some help.

Acknowledgements

We would like to thank Toby Walsh for his important comments on this work and on the HR project. We would like to thank Pat Langley and Raúl Valdés-Pérez for sending us survey papers on machine discovery programs. We would also like to thank Susan Epstein for her comments explaining aspects of the GT program, Jan Żytkow for his comments on the system by Bagai et al and Craig Larson for helping us understand the Graffiti program. This work is supported by EPSRC research grant GR/L 11724.

References

R Bagai, V Shanbhogue, J Żytkow, and S Chou. Automatic theorem generation in plane geometry. In *LNAI 689*. Springer Verlag, 1993.

A Bundy, S Colton, and T Walsh. HR - automatic concept formation in finite algebras. Technical Report 920, (Presented at the Machine Discovery Workshop at ECAI 98) Department of Artificial Intelligence, University of Edinburgh, 1998.

S Chou. *Mechanical Theorem Proving*. D. Reidel Publishing Company, Dordrecht, Netherlands, 1984.

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

S Colton and A Bundy. HR: Automatic concept formation in pure mathematics. In *Proceedings of IJCAI*, 1999.

R Davis and D Lenat. *Knowledge-Based Systems in Artificial Intelligence*. McGraw-Hill Advanced Computer Science Series, 1982.

S Epstein. *Knowledge Representation in Mathematics: A Case Study in Graph Theory*. PhD thesis, Department of Computer Science, Rutgers University, 1983.

S Epstein. On the discovery of mathematical theorems. In *IJCAI Proceedings*, 1987.

S Epstein. Learning and discovery: One system's search for mathematical knowledge. *Computational Intelligence*, 4(1):42–53, 1988.

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

S Fajtlowicz. The writing on the wall. Unpublished preprint, available from http://math.uh.edu/ clarson/, 1999.

P Langley. The computer-aided discovery of scientific knowledge. In *Proceedings of the first international conference on discovery science*, 1998.

P Langley, H Simon, G Bradshaw, and J Żytkow. *Scientific Discovery - Computational Explorations of the Creative Processes*. MIT Press, 1987.

C Larson. Intelligent machinery and discovery in mathematics. Unpublished preprint, available from http://math.uh.edu/ clarson/, 1999.

D Lenat. *AM: An artificial intelligence approach to discovery in mathematics*. PhD thesis, Stanford University, 1976.

W McCune. The OTTER user's guide. Technical Report ANL/90/9, ANL, 1990.

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

G Ritchie and F Hanna. AM: A case study in methodology. *Artificial Intelligence*, 23, 1984.

N Sloane. *The Online Encyclopedia of Integer Sequences*. http://www.research.att.com/~njas /sequences, 1998.

R Valdés-Pérez. Principles of human computer collaboration for knowledge discovery in science. *Artificial Intelligence*, 107(2), 1999.