This paper appeared in 1988 in *Computational Intelligence* 4 (1): 42-53.

Learning and Discovery:

One System's Search for Mathematical Knowledge

Susan L. Epstein
Department of Computer Science
Hunter College and The Graduate School
The City University of New York
New York, NY 10021

Keywords: discovery, machine learning, knowledge representation, concept definition, mathematics, graph theory

Abstract

The Graph Theorist, GT, is a system that performs mathematical research in graph theory. From the definitions in its input knowledge base, GT constructs examples of mathematical concepts, conjectures and proves mathematical theorems about concepts, and discovers new concepts. Discovery is driven both by examples and by definitional form. The discovery processes construct a semantic net that links all of GT's concepts together.

Each definition is an algebraic expression whose semantic interpretation is a stylized algorithm to generate a class of graphs correctly and completely. From a knowledge base of these concept definitions, GT is able to conjecture and prove such theorems as "The set of acyclic, connected graphs is precisely the set of trees" and "There is no odd-regular graph on an odd number of vertices." GT explores new concepts either to develop an area of knowledge or to link a newly-acquired concept into a pre-existing knowledge base. New concepts arise from the specialization of an existing concept, the generalization of an existing concept, and the merger of two or more existing concepts. From an initial knowledge base containing only the definition of "graph," GT discovers such concepts as acyclic graphs, connected graphs and bipartite graphs.

The Graph Theorist, GT, is a knowledge-intensive, domain-specific learning system (Michalski 1986) that uses algorithmic class descriptions to discover new mathematical concepts and relations among them. GT is a based upon a set of powerful representation languages for object classes discussed and defined formally in (Epstein 1983). A variety of well-defined operations can be coerced from these languages. In particular they can be used to:

- generate correct examples of any class
- test whether or not an object belongs in a particular class
- reason about relations among classes

The representation of a class in any of these languages is an algebraic definition. Each definition has a semantic interpretation as a stylized algorithm that defines the class by generating it correctly and completely. GT incorporates such definitions as slots in its frame representation for a graph property. The entire frame represents a concept.

GT operates either independently or under interactive guidance. It generates correct examples of any of its concepts, constructs new concepts, and conjectures and proves relations among concepts. The first section of this paper places GT in the context of previous work in machine learning and discovery. The second section introduces a set of knowledge representation languages, and the transition from the languages to algorithms and then to the concepts for GT's knowledge base. The next sections describe the discovery of relations between concepts in GT and the ways GT constructs new concepts. Subsequent sections summarize GT's discovery modes, and its results and significance. The final section presents some open questions and plans for future work.

1. Context and Background

1.1 The Representation of Objects and Classes

Given a set of objects in a domain, the task of inductively inferring their natural classes and the relations among those classes may be categorized as learning by discovery (Carbonell, Michalski, and Mitchell 1983). When the objects are many and highly detailed, search over the domain usually requires heuristic guidance.

The most common heuristic is to select certain attributes of the objects as relevant to the search and then to summarize each object by describing certain of its attribute values in a feature vector. If an object exists in the real world, such a description must be a generalization of it, since the description is certain to omit some presumably irrelevant attributes, such as surface temperature, molecular structure or country of origin. Even if an object does not have a counterpart in the real world, a feature vector representation may impose a loss of information. Consider, for example, a graph G on 25 vertices with 237 edges. G is a theoretical object consisting by definition of only the list of its vertices and the list of its edges. A feature vector representation of G, such as <25, 237, T, F,...>, might detail the number of G's vertices, the number of its edges, whether or not it is connected, whether or not it is planar, and so on. No matter how long the feature vector, if G's only defining characteristics (exactly what are the vertices and edges) are omitted, in a system using the feature vector representation, G *becomes* its attribute values. If a system represents G as <25, 237, T, F,...> and subsequently needs to calculate the value of some graph property not among the listed features, the computation, although well-defined on G, may be impossible from <25, 237, T, F,...>.

The loss of detail incurred by a feature vector representation is expected to be justified by an ability to support the efficient computation of simple descriptions while maximizing inter-class differences (Michalski and Stepp 1983). Once such rules for classes have been inferred, however, the next task may be to infer the relations among the classes. There is no guarantee that the features that were relevant to class identification will be equally useful in reasoning about the classes. For example, the food chain domain of (Nordhausen 1986) contains facts about a variety of birds; within it, reasoning about hierarchical relations requires the definition of new attributes, such as the size of what a bird eats, to infer a classification tree. There are, of course, underlying structural similarities between hawks and owls that would also distinguish them from songbirds, but those structural details were lost when the classes, hawks and owls, were represented by their features. Thus an approximation must be discovered in place of the discarded details about structural similarities.

Constraining the solution space to the vocabulary of the original attributes has been repeatedly identified as a severe limitation (for example, (Stepp and Michalski 1986)), and there has been much recent interest in expanding the expressive power of feature-based descriptions. The BACON programs (Langley, Bradshaw, and Simon 1983; Langley, Zytkow, Simon, and Bradshaw 1986) postulate and calculate new numerical attributes from the original ones. OPUS (Nordhausen 1986) postulates and calculates new attributes as relational combinations of the original ones. Other expansions are formed in STAHL and GLAUBER (Langley, Zytkow, Simon, and Bradshaw 1986), in (Lee and Ray 1986), and in (Laird 1986).

Even when the representation of a concept is not explicitly a feature vector, inadequacies in the description language present research challenges. STABB (Utgoff 1986) creates disjunctions of existing attributes to expand LEX's solution space. CLUSTER/S (Stepp and Michalski 1986) uses either numerical combinations or logical operators to create new attributes. PI (Thagard and Holyoak 1985) combines frames containing production rules. Other efforts (Emde, Habel, and Rollinger 1983) use mathematical properties, such as transitivity, to combine relational attributes.

GT takes an alternative approach to the detection of relations among object classes. It presupposes a concise class description that, unlike feature vectors, has no loss of detail and yet expedites search. The commonality among objects in a class is based, not on a set of attributes, but on inherent structural similarities in the objects' construction. The class description is really a non-deterministic generation algorithm that produces the object class. The structural uniformity imposed upon the algorithms themselves then supports discovery of relations between classes. Of course, not every domain encourages such class descriptions or expects a rich set of relations among the resultant classes.

1.2 Mathematics as a Domain for Discovery

Machine learning uses both data and theories to drive its models of scientific discovery. Much important work in machine learning (for example, (Laird 1986) (Langley, Bradshaw, and Simon 1983; Langley, Zytkow, Simon, and Bradshaw 1986; Lee and Ray 1986; Michalski and Stepp 1981; Michalski and Stepp 1983; Shapiro 1981; Stepp and Michalski 1986)constructs concepts to explain empirical data, data either input or encountered in the course of task execution. Such data-driven discovery is designed to infer explanations for factual input and is based upon observation and inductive inference, in domains where partial information and real-world noise are the norm.

The theoretical sciences concern themselves with highly detailed objects, objects already bound into classes by strong structural similarities. Mathematics, in particular, has many classes of objects already clearly delineated by definitions. In modeling such domains, the origin of these classes (what (Michalski 1986) calls *constructive induction*) and the relations among them should be the focus of attention. In theory-driven discovery, search heuristics postulate statements and then run experiments to validate those statements. Mathematics is a scientific domain with noise-free data and clear criteria (e.g., proofs and counterexamples) to validate and invalidate theories. For centuries, the clarity and certainty of the inherent relations among its classes have lured mathematicians with prospects of discovery.

One outstanding example of computer discovery in mathematics was Lenat's AM (Lenat 1976). AM began with a set of 115 frames, each representing a mathematical concept. A frame consisted of fixed slots containing information on examples, hierarchical pointers, conjectures about such pointers and heuristics attached to the concept. The definitive slot in an AM concept, however, was the predicate in the definition, a LISP testing algorithm that ruled on whether or not an object was a member of the class. Some of AM's initial (i.e., input) concepts were for activities defined on those classes, such as set-insert or list-intersect. Examples for a class were generated by applying seemingly appropriate operators. For example, if an operator f were defined on the classes A and B with $f{:}A \rightarrow B$, and if C were a subset of B, AM might have selected some examples $a_1, a_2, \ldots, a_n$ of A, constructed $f(a_1), f(a_2), \ldots, f(a_n)$, all of which would have belonged to B, and then examined them to see if they were indeed examples of C, or merely in B - C. This uncertainty in the generation of examples, and the need to check them with the testing algorithm, occurred because an AM definition for a class of mathematical objects could only confirm membership in a class, not produce elements of it.

AM explored its fundamental frames under the guidance of 243 heuristic rules. It generated (and then checked) examples of the concepts, observed simple statistical regularities in their classification, conjectured about relations among them, and defined new frames (concepts) for subsequent exploration. AM developed a new concept by small modifications to the LISP function that served as the definition of an existing concept. Thus the LISP code for TEST, to check whether or not an example was a member of some class D, would be modified to create a new algorithm TEST' for membership in a (presumably) different class D'. The definition of a new AM concept did not differ greatly from its parent concept; only small changes could be guaranteed to produce executable code, and AM was, after all, not intended to be an expert in automated programming.

Discovery in AM was inductive inference, driven, not only by empirical evidence, but also by the heuristic rules that originated example generation and evaluated the results. This discovery was driven both by data (the generated examples and the initial concepts) and by theory (the heuristics for mathematical research behavior). Beginning only with set-theoretic concepts, such as bag and union, AM included among its discoveries natural numbers, primes, and the fundamental theorem of arithmetic. Lenat believed that AM did not go on to explore other areas of mathematics because its heuristics (i.e., theory) were not as relevant to other subject matter. (This motivated his subsequent work on EURISKO, a system that discovers new heuristics as well.) Ostensibly, AM’s attention during search was focused by what tasks AM had recently completed and by each concept’s worth, a numerical measure of “interestingness.” When Lenat experimented with variations in the originally-input worth values for the first 115 concepts, AM did not pursue radically different goals. Initially this was attributed to the strength of the heuristics. Only later (Lenat and Brown 1984; Ritchie and Hanna 1984) was it generally recognized that the path of discovery was pronouncedly affected by the programming language itself. AM developed new testing algorithms by modifications to other algorithms, i.e., to LISP code. Because the syntax of the class definitions carried a semantic bias toward certain structures (such as ordered sets), AM was fated to explore and discover in a somewhat predetermined fashion.

Despite its attraction to LISP-like structures, AM was a successful program. Indeed, AM may be said to model some intuition on the part of the research mathematician as to a representation language (LISP) well-suited for exploration in a particular direction (number theory). AM certainly suggests that, if knowledge about a domain can be semantically encoded into the class definition, then it can be harnessed to drive mathematical discovery.

GT encodes the semantics of graph theory into class definitions in a more transparent and flexible fashion, one that supports both inductive and deductive reasoning. These semantics subsequently motivate both conjecture and proof, generate guaranteed examples of a class, and permit the introduction of new descriptions into class definitions. (Recall that AM was never able to prove its conjectures, to generate new examples certain to be in a specified class, or to introduce radically new descriptions into class definitions.) GT's discovery process is highly-constrained exhaustive search. Although the space of all possible classes of graphs is large, the representation language is able to control and focus exploration through it in a variety of ways. When it begins with an extremely general definition and explores specialized versions of it fairly exhaustively, GT is similar to META-DENDRAL (Buchanan and Mitchell 1978). Both programs survive a generate-and-test strategy because their representations encourage what Michalski (Michalski 1983) calls *conceptual data analysis*.

2. Concept Description in GT

GT derives much of its power from a set of representation languages whose theoretical formulation is detailed rigorously in (Epstein 1983) and summarized in (Epstein 1987). The treatment of the representation here is informal and describes only selected, implemented segments of the theory. For example, GT currently supports only undirected, unlabeled graphs, but coding provisions have been made for directed and labeled graphs, and the theoretical framework supports them.

Figure 1 portrays the development of a GT concept from the representation languages. An expression in one of these languages is an algebraic definition for a class of graphs. Any such expression has a semantic interpretation as a stylized algorithm that defines the class by generating it correctly and completely. One expression fills the definition slot in each GT concept frame. This section provides some fundamental graph terminology and describes the

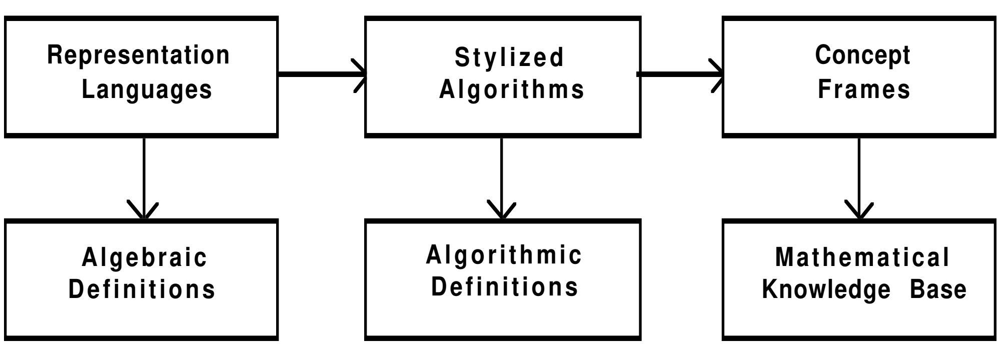

Figure 1: The Formulation of a GT Concept

algebraic definitions, their uniform semantic interpretation, and the concept frames.

Let V be an arbitrary, finite set of elements (*vertices*), and let E be any subset (*edges*) of the Cartesian product $V \times V$. Then the ordered pair G=<V,E> is said to be a *finite graph* or, more simply, a *graph*. If $|V| = n$ and $E = \{(x,y) \mid x,y \in V, x \neq y\}$ then G=<V,E> is said to be *complete* and is denoted here as K_n. If $x \in V$ but there is no edge (x,y) or (y,x) in E, x is said to be *isolated*.

Let U be the universe of all finite graphs. Then any subset P of U is said to designate a *graph property p* and, for G in P, G is said to *have property p.* (The distinction between the property p and the class P is syntactic, not semantic. The context will dictate which is used.) Any algorithmic definition of the graph property p must specify the set P precisely. In particular, if an algorithm claims to generate P, that algorithm must be both *correct* (i.e., every generated graph must be in P) and *complete* (i.e., for each graph in P there must be a finite sequence of steps executed by the algorithm with final output G).

In GT, a *concept* is a frame representing a graph property and knowledge associated with it. (This representation was inspired by Michener's work (Michener 1978).) A slightly-edited example of an initial GT frame for the concept ACYCLIC appears in Figure 2. The slots of the frame include a list of examples, knowledge about hierarchical relations with other concepts, historical information on the ways the concept has been manipulated, and a description of the origin of the property. (Entries of "nil" for relations are statements of partial knowledge, to be read as "none discovered yet.") The frame also includes a definition of the graph property in a

Property name	ACYCLIC
Number-of-seeds	1
Seed-set	$\{K_1\}$
Function	$A_x + A_{yz}A_z$
Sigma	$y \in V$, $x,z \notin V$
Origin	input
Example-list	(K_1)
Extremal-cases	(K_1)
Delta-pairs	((1 0) (1 1))
Subsumes	nil
Cannot-be-shown-to-subsume	nil
Subsumed-by	nil
Cannot-be-shown-subsumed-by	nil
Is-equivalent-to	nil
Cannot-be-shown-equivalent-to	nil
Merger-created-with	nil
Merger-explored-with	nil
Has-been-generalized	nil
Has-been-specialized	nil
Variables-have-been-identified	nil
Iteration-has-been-forced	nil

Figure 2: Initial Representation of ACYCLIC

Table 1: Primitive GT Operators

Symbol	Application	Interpretation
A_x	Operator	Add vertex x to the graph: $V \leftarrow V \cup \{x\}$
A_{xy}	Operator	Add edge xy to the graph: $E \leftarrow E \cup \{xy\}$
D_x	Operator	Delete vertex x from the graph: $V \leftarrow V - \{x\}$
D_{xy}	Operator	Delete edge xy from the graph: $E \leftarrow E - \{xy\}$

specific, three-part formulation.

In GT, a *definition* of a graph property is an ordered triple $<f,S,\sigma>$. S is the *seed set*, a set of one or more minimal graphs (*seeds*), each of which has the property in question. (Typically the seed set is finite and GT lists its elements.) The seed set in the example of Figure 2 for ACYCLIC contains only a single graph, K_1. The *operator* f in the definition describes the way(s) any graph with the given property may be transformed to construct another graph with the same property. An operator in GT is built from the set of primitives listed in Table 1. There is a primitive to add the vertex x (A_x), to add the edge between x and y (A_{xy}), to delete the vertex x (D_x), and to delete the edge between x and y (D_{xy}). These primitives may be concatenated into *terms*, (such as "$A_{yz}A_z$") to denote sequential operation from right to left. Terms may be summed (as in "$A_x + A_{yz}A_z$") to represent alternative actions. Thus the operator $A_x+A_{yz}A_z$ for ACYCLIC is read "either add a vertex x or else add a vertex z and then an edge from y to z." The *selector* σ in the definition describes the restrictions for binding the variables appearing in the operator f to the vertices and edges in a graph. Selector descriptions for vertices and edges appear in Table 2. The current valid selector descriptions for a vertex describe whether or not it is in the graph, its distinctness from another specific vertex symbol, its degree (number of neighbors), and whether or not its degree is the maximum among the degrees of all the vertices in the graph. The current valid edge selector descriptions in GT are of two kinds: whether or not the edge is in the graph, and whether its endpoints are distinct. Selector descriptions may be empty, i.e., need not

Table 2: GT Selector Descriptions

Symbol	Application	Interpretation
$x \in V$	Selector	x is a vertex in the graph.
$x \notin V$	Selector	x is a vertex not in the graph.
$x \neq y$	Selector	x and y are distinct vertices.
$d(x) = n$	Selector	x's degree is n, a non-negative integer.
$d(x) = max$	Selector	x's degree is the largest in the graph.
$xy \in E$	Selector	xy is an edge in the graph.
$xy \notin E$	Selector	xy is an edge not in the graph.

constrain binding at all. In the example of Figure 2, the selector for ACYCLIC is read "where y is in the vertex set, and x and z are not in the vertex set."

The semantic interpretation of such a three-part definition for a graph property p is a single, uniform algorithm called a *p-generator*. A p-generator capitalizes on the underlying commonality of its class, the view of the set P as one or more prototypes (seeds) that can be methodically transformed (under f and σ) to produce exactly those graphs in the class. The p-generator may be thought of as an automaton that is started by the input of any graph in its seed set S. ACYCLIC, for example, would permit only K_1. The P-generator then iterates an undetermined number of times. On each iteration the selector σ chooses vertices and/or edges with respect to the current graph G, and then the operator modifies G, using those choices to produce a new G. ACYCLIC, on each iteration, either adds a new vertex x to the graph, or adds a new vertex z and an edge from an old vertex y to z.

Thus the algorithm for generating the class P of graphs is

```
Accept G in S
Output G
Until s fails do
    G <- fs(G)
    Output G
Halt
```

Under all possible initial choices from S and all possible iterations of f subject to σ, the output of this algorithm is precisely P, that is, if the superscript i denotes "iterate i times,"

$$P = \bigcup_i (f\sigma)^i S$$

The graphs in Figure 3 illustrate several possible iterations of the definition of ACYCLIC; each pictured graph is output by the algorithm and is acyclic. The definition generates the infinite class of acyclic graphs; it will never halt because bindings for the variables in σ can be found on each iteration.

The content of the following three general texts is taken as *graph theory*: (Ore 1962), a classical development in elegant mathematical fashion, (Harary 1972), a broad overview of

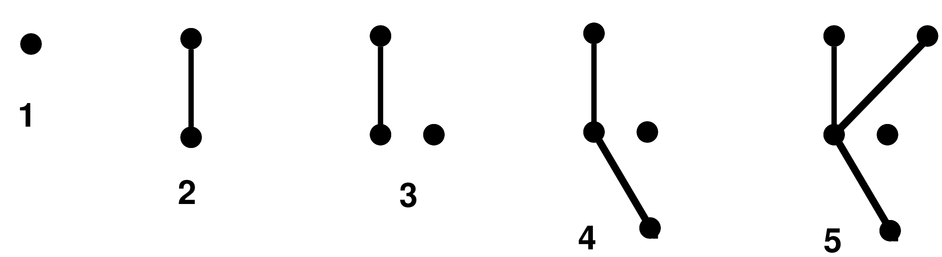

Figure 3: GT Iterations of ACYCLIC

Table 3: Graph Properties Representable as P-Generators

graph	graph on even number of vertices
edgeless graph	graph on odd number of vertices
connected graph	graph with even number of edges
biconnected graph	graph with odd number of edges
acyclic graph	graph of minimum degree k
k-connected graph	graph with k components
tree	graph with counted vertices
loopfree graph	graph with counted edges
chain	graph with calculated maximum degree
cycle	bipartite graph
star	complete bipartite graph
wheel	k-vertex-covered graph
complete graph	k-independent graph
Eulerian graph	k-colored graph
graph with n vertices	k-chromatic graph
graph with m edges	graph with vertex covering number k
pinwheel	graph with circumference k
non-planar graph	graph with edge covering number k
k-factorable	graph with a k-factor
even-regular graph	Hamiltonian graph
odd-regular graph	planar graph

topics presented as definitions and theorems, and (Bondy and Murty 1976), an algorithmic approach. There is some evidence that p-generators may exist for every P in U, or at least for every interesting P in graph theory. At this writing, more than 40 properties of varying difficulty have been selected from the three benchmark texts and described correctly and completely as p-generators (Epstein 1983). They appear in Table 3. Three graph properties (self-complementary, uniquely k-colorable, and k-edge-colorable) have been expressed and proved correct but lack completeness proofs, although no counterexamples are known. Current research suggests that a more extensive knowledge of graph theory would resolve such difficulties.

The use of p-generators as property definitions entails several kinds of non-determinism. Any graph in the seed set is an acceptable input; any binding satisfying σ is valid; any term in f suffices for an iteration. In addition, many different sequences of iterations will construct isomorphic graphs, and more than one definition may be written for certain properties. (GT stores equivalent definitions as separate properties with connecting links.) This ostensible indefiniteness and redundancy is tolerated because the property definitions preserve detail in a concise and flexible format.

3. Relations between Concepts in GT

As Michalski has observed (Michalski 1983), inductive inference from examples does not preserve truth but only falsity. Although research mathematicians devote much time to example generation, and infer conjectures about relations among mathematical ideas based on these examples, inductive inference is only a tool. Rarely is a conjecture considered a result worthy of publication, and then only when extensive attempts at proof and disproof have failed. Mathematicians prefer to explore in the context of certainty; for them a conjecture should be proved or disproved relatively soon after it arises. GT, therefore, constructs both conjectures and proofs.

Graph theory, as it appears in the three benchmark texts cited above, is primarily about graph properties and the relations among them. Conjectures and theorems in graph theory frequently take one of the following forms:

- TYPE 1: If a graph has property p, then it has property q.
- TYPE 2: A graph has property p if and only if it has property q.
- TYPE 3: If a graph has property p and property q, then it has property r.
- TYPE 4: It is not possible for a graph to have both property p and property q.

GT has two fundamental proof procedures for manipulating graph properties to prove such theorems. The first procedure tests for subsumption. Property p for class P *subsumes* property q for class Q if and only if $P \supseteq Q$, i.e., every graph with property q also has property p, so that q is a special case of p. Theorem type 1 is a statement that q subsumes p. Theorem type 2 is a statement that p and q are equivalent, i.e., that p subsumes q and q subsumes p. The second procedure constructs mergers. The *merger* of a property p for class P with a property q for class Q results in a new property representing $P \cap Q$, the set of graphs with both properties. Theorem type 3 is a statement that property r subsumes the merger of p and q. Theorem type 4 is a statement that the merger of p and q is empty, i.e., that no graph can have both properties simultaneously. The remainder of this section describes how subsumption is tested and how mergers are constructed using the $\langle f,S,\sigma \rangle$ representation.

3.1 Proof of Subsumption

At this writing, GT has only one method of proving subsumption. Given property $p_1 = \langle f_1,S_1,\sigma_1 \rangle$, property $p_2 = \langle f_2,S_2,\sigma_2 \rangle$, and a conjecture that p_1 subsumes p_2, GT attempts to show that:

- f_2 is subsumed by f_1, that is, f_2 is a special case of f_1. (Extended definitions for operator subsumption appear in (Epstein 1983).)
- Every graph in S_2 has property p_1.
- σ_2 is subsumed by σ_1, that is, σ_2 is more restrictive than σ_1. (Extended definitions for selector subsumption appear in (Epstein 1983).)

Because there are usually only a few known seeds, GT checks the list of examples for p_1 against S_2. If any graph G in S_2 is not known to have p_1, GT generates a limited number of new examples of p_1 and searches for G there. Because seed graphs are extremal cases, and because a natural metric exists on most GT definitions, the search is readily controlled and usually successful. (Alternative techniques exist for infinite seed sets and certain other situations.) Matching for the subsumption testing of the operators and selectors is done by a recursive backtracking algorithm that generates a restricted set of candidates.

The following example of subsumption testing illustrates the subsumption procedure. For

$$\text{ACYCLIC} = \langle A_x + A_{yz}A_z, \{K_1\}, [y \in V, x,z \notin V]\rangle$$

and

$$\text{TREE} = \langle A_{pq}A_q, \{K_1\}, [p \in V, q \notin V]\rangle$$

and the conjecture "ACYCLIC subsumes TREE," GT must show that K_1, the seed for TREE, is an acyclic graph and also that, under some matching, the ACYCLIC operator "covers" the TREE operator while satisfying the TREE selection constraints. First, K_1 is on the list of acyclic graphs because it is the seed for ACYCLIC. Second, the matcher notes that every term in the TREE operator $A_{pq}A_q$ (there is only one in this example) is covered by some term, namely $A_{yz}A_z$, in the ACYCLIC operator. Finally, the matcher observes that under the matching of p with y and q with z, the selector constraints (that p is in V and q is not) are enforced. Thus GT proves that ACYCLIC subsumes TREE or, more formally, "Every tree is an acyclic graph."

3.2 Proofs Involving Mergers

GT currently has four algorithms for merger. Given property $p_1 = \langle f_1,S_1,\sigma_1\rangle$ and property $p_2 = \langle f_2,S_2,\sigma_2\rangle$, GT attempts to construct the merger $p = \langle f,S,\sigma\rangle$ of p_1 and p_2. The first three algorithms are fairly straightforward:

- If p_1 subsumes p_2, the merger is simply p_2. For example, the merger of ACYCLIC and TREE is simply TREE.
- When f_1 subsumes f_2 and every seed in S_2 has property p_1, the merger is $\langle f_2,S_2,\sigma\rangle$, where σ is σ_1 and σ_2, but eliminates any references to variables not in f_2. For example, the merger of STAR = $\langle A_{xy}A_y, \{K_{1,2}\}, [x \in V, y \notin V, d(x) = \max]\rangle$ and ACYCLIC is simply STAR.
- When f_1 subsumes f_2, σ_1 subsumes σ_2 , and S is non-empty, the merger is $\langle f_2,S,\sigma_2\rangle$, where $S = \{G \mid G \in S_2 \cap P_1\} \cup \{G \mid G \in S_1 \cap P_2\}$. For example, the merger of

$p_1 = \langle A_x + A_{yz}A_z, \{K_1, K_4\}, [x \notin V, y \in V]\rangle$
with $p_2 = \langle A_{rs}A_s, \{K_1, K_3\}, [r \in V, s \notin V]\rangle$ where clearly $K_3 \in P_1$ but $K_4 \notin P_2$ is
$p = \langle A_{uv}A_v, \{K_1, K_3\}, [u \in V, v \notin V]\rangle$ based on matching y with r as u and z with s as v.

The fourth, and most interesting, of GT's merger algorithms deals with the cases that do not fit these categories. Let n be the number of vertices in a graph, and m be the number of edges. Each iteration of the p-generator effects a change (Δn) in the number of vertices and a change (Δm) in the number of edges. GT calculates the Δn and Δm values for each term in the properties to be merged. A *delta pair* (Δn, Δm) is the ordered pair of the changes for a term in a property; it captures some aspects of the minimal effect of one iteration of a p-generator. For example, the only delta pair for

$$\text{TREE} = \langle A_{pq}A_q, \{K_1\}, [p \in V, q \notin V]\rangle$$

is (1,1), meaning that on each iteration one vertex and one edge are added to the graph. For

$$\text{ODD-VERTICES} = \langle A_rA_s + A_{tu}, \{K_1\}, [r,s \notin V, t,u \in V, r \neq s]\rangle$$

the delta pairs are (2,0) and (0,1). For a merger, GT seeks a minimal positive integer solution to that system of equations which asserts that some number of repetitions of the delta pair for each term in one property is equivalent to some repetitions of the delta pairs in the second property. In the example, let α represent the number of applications of the single term in TREE, let β represent the number of applications of the first term in ODD-VERTICES, and let μ represent the number of applications of the second. GT seeks the positive minimal integer solutions to:

$$1\alpha = 2\beta + 0\mu \ (\Delta n)$$
$$1\alpha = 0\beta + 1\mu \ (\Delta m)$$

The answer, $\alpha = \mu = 2$ and $\beta = 1$, indicates that in the merger both Δn and Δm will be 2. Each of the properties is specialized by the repetition of appropriate terms to meet these requirements:

$$\text{TREE}' = \langle A_{xy}A_y\ A_{uv}A_v, \{K_1\}, [x,u \in V, y,v \notin V]\rangle$$
$$\text{ODD-VERTICES}' = \langle A_rA_sA_{tz}A_{pq}, \{K_1\}, [r,s \notin V, t,z,p,q \in V, r \neq s]\rangle$$

When GT attempts a merger of TREE' and ODD-VERTICES' it discovers that the first is really a special case of the second, under the matching of r and z with y, s and q with v, t with x, and p with u. (An extremely limited form of commutativity is used here to shift operators of the form A_x to the right when the vertex does not appear elsewhere in the term.) Thus the merger is

$$\text{ODD-TREE} = \langle A_{xy}A_y\ A_{uv}A_v, \{K_1\}, [x,u \in V, y,v \notin V, y \neq v]\rangle$$

Upon inspection, this property is clearly correct and complete, with $\Delta n = \Delta m = 2$.

GT has discovered, among other merger properties, TREE as ACYCLIC merged with CONNECTED, and COMPLETE-BIPARTITE as COMPLETE merged with BIPARTITE.

Some of the most interesting of GT's proofs are merger failures. Consider, for example, GT's discovery that a graph that is odd-regular (every vertex of degree d, and d is odd) cannot have an odd number of vertices. GT looks first for a common seed. Since none is evident, GT generates some examples to expand its list of graphs with an odd number of vertices, hoping to find one which it recognizes as odd-regular. When this effort fails, GT considers the possibility that there is no common seed, and examines the changes to m and n wrought by the operators. GT recognizes that ODD-NUMBER-OF-VERTICES begins with one vertex and adds two vertices at a time, so that n is always odd. GT recognizes that ODD-REGULAR begins with an even number of vertices (the seed is K_2) and adds an even number of vertices at a time, so that n is always even. This disparity is the reason GT gives in its proof: there can never be a seed for the merger, and thus the property has no example, i.e., is impossible.

4. Construction of New Concepts in GT

GT definitions transparently display both the changes they force upon objects (in the operator f) and the preconditions they require (in the selector σ). This separation encourages the development of completely and correctly defined new concepts whose relations to their parent concepts do not require proof. GT currently has three methods for constructing new concepts from known ones: specialization, generalization, and merger.

4.1 New Concepts Discovered by Specialization

When property p_1 subsumes property p_2, every graph with property p_2 also has property p_1. Thus p_2 may be viewed as a specialization of p_1. To specialize from a definition of the form $<f,S,\sigma>$, GT performs one of the following actions:

- constrain the seed set
- constrain the operator
- constrain the selector

When GT discovers property p_2 as a specialization of property p_1, the facts that P_2 is a subset of P_1, and P_1 a superset of P_2, are recorded in GT's knowledge base. The specialization techniques described here are those currently implemented; others are under development.

GT constrains a seed set by using a proper subset of it. Consider once again the property

$$p_1 = <A_x + A_{yz}A_z, \{K_1, K_4\}, [x \notin V, y \in V]>$$

The definition p_1 begins with a complete graph, either K_1 or K_4. On any iteration, p either adds an isolated vertex x to the graph or adds a vertex z and an edge yz from a vertex y in the graph to z. One specialization of p_1 is

$$p_3 = <A_x + A_{yz}A_z, \{K_4\}, [x \notin V, y \in V]>$$

created by eliminating K_1 from the seed set of p_1. Every graph generated by p_3 begins with a seed from p_1 (namely, K_4), iterates according to the definition of p_1, and, therefore, has property p_1. There are, however, graphs (for example, K_2 and K_3) that are in P_1 but not in P_3. Thus p_3 is a proper subset of p_1.

An operator may be constrained in two ways. First, a term may be eliminated from the operator (with irrelevant constraints removed from the selector). For example,

$$p_4 = \langle A_{yz}A_z, \{K_1, K_4\}, [y \in V]\rangle$$

constructs only the connected p_1 graphs. One may prove p_4 a specialization of p_1 by the argument used for p_3 above. Second and less obviously, recall that a p-generator assumes iteration. Thus any forced repetition of terms from the operator forms a special case of the operator. (The selector requires readily-computable additions.) Consider

$p_5 = \langle A_xA_w + A_tA_{yz}A_z, \{K_1, K_4\}, [x,w,t \notin V, y \in V]\rangle$

Property p_5 adds either a pair of isolated vertices or an edge and two vertices (at least one of which is new) on each iteration. Property p_5 begins with a seed from p_1, and each of its terms is equivalent to a finite number of iterations of p_1; therefore, p_5 is a specialization of p_1.

GT constrains the selector of a graph property by making the binding restrictions more detailed, either by the addition of a constraint or the identification of variables. As an example of the first, consider

$$p_6 = \langle A_x + A_{yz}A_z, \{K_1, K_4\}, [x,z \notin V, y \in V]\rangle$$

Using the argument employed for p_3, p_6 is seen to be a specialization of p_1; P_6 is that subset of P_1 that is acyclic everywhere except possibly in a single K_4 subgraph. Additional constraints must always be consistent with the definition of a graph, and never obviously make binding impossible. (For example, $x \in V$ would not be added when the selector already specifies $x \notin V$). For an example of the second selector specialization, identification of variables, consider:

$$p_7 = \langle A_x + A_{yy}A_y, \{K_1, K_4\}, [x \notin V, y \in V]\rangle$$

Here GT has selected two variables in p_1, y and z, whose σ-descriptions do not contradict each other, and has made them identical.

Figure 4 shows how GT applies specialization to discover new concepts in graph theory. Initially, the knowledge base consists only of line 1, the p-generator for all finite graphs. The more interesting properties have been selected for the figure from the trace, and some have been renamed for the figure. First, the general definition of a graph is specialized by forced repetition

	Property	**Origin**
1	IS-A-GRAPH: $\langle A_{xy} + A_z, \{K_1\}, [x,y \in V]\rangle$	Given
2	PROPERTY-4: $\langle A_{xy}A_z, \{K_1\}, [x,y \in V]\rangle$	Forced repetition in (1)
3	PROPERTY-6: $\langle A_{xy}A_z + A_w, \{K_1\}, [x,y \in V]\rangle$	Forced repetition in (1)
4	PROPERTY-14: $\langle A_{xz}A_z, \{K_1\}, [x \in V]\rangle$	Identify variables in (2)
5	TREE: $\langle A_{xz}A_z, \{K_1\}, [x \in V, z \notin V]\rangle$	Restrict binding in (4)
6	CONNECTED: $\langle A_{xz}A_z, \{K_1\}, [x \in V, z \notin V]\rangle$	Restrict binding in (4)
7	PROPERTY-30: $\langle A_{xz}A_z + A_w, \{K_1\}, [x \in V]\rangle$	Identify variables in (3)
8	ACYCLIC: $\langle A_{xz}A_z + A_w, \{K_1\}, [x \in V, z \notin V]\rangle$	Restrict binding in (7)

Figure 4: GT Discovers New Concepts: A Trace

to produce PROPERTY-4 and PROPERTY-6. The identification of variables in PROPERTY-4 produces PROPERTY-14. Constraining σ in PROPERTY-14 produces TREE and CONNECTED. On another exploration branch, the identification of variables in PROPERTY-6 produces PROPERTY-30. Constraining σ in PROPERTY-30 produces ACYCLIC.

A *loop* is an edge from a vertex to itself. In another discovery sequence, GT constructs

$$\text{BIPARTITE} = \langle A_x{+}A_{yy}A_y{+}A_{wz}, \{K_1\}, [x,y \notin V, w,z \in V, ww \in E, zz \notin E]\rangle$$

For BIPARTITE, GT has partitioned the vertices of the graph into two sets, one with loops and one without; edges are drawn only between one vertex with a loop and one without.

4.2 New Concepts Discovered by Generalization

A property p_2 is said to be a generalization of a property p_1 if and only if P_1 is a subset of P_2, i.e., every graph with property p_1 also has property p_2. Because "p_2 is a generalization of p_1" is equivalent to "p_1 is a specialization of p_2," the construction of generalizations is fairly obvious from the preceding discussion To generalize a concept, GT may:

- expand the seed set
- expand the operator
- relax the selector

To expand a seed set, GT adds another graph or set of graphs to it. To expand an operator, GT adds new terms or splits existing ones (the inverse of forced iteration). To relax the selector, GT removes details from the binding restrictions in σ. Each of the examples of specialization in Section 4.1 may be read, in reverse, as an example of generalization. When GT discovers a new concept p_2 as a generalization of a concept p_1, the facts that P_2 is a superset of P_1, and P_1 a subset of P_2, are recorded in the knowledge base.

Why would GT need to know how to generalize at all? GT models a variety of research behaviors, one of which is the appropriate insertion of new information into a pre-existing knowledge base. A new property is generalized until it can be linked into GT's relational hierarchy. For example, when given

$$\text{STAR} = \langle A_{xy}A_y, \{K_{1,2}\}, [x \in V, y \notin V, d(x) = \max]\rangle$$

the concept is generalized until it is identified with one already in the knowledge base. Directed to relax STAR's binding constraints, GT produces three new property definitions, one of which differs from TREE only in its seed. When directed to relax the constraints once again, GT produces two definitions, one of which differs from line 4 of Figure 4 only in its seed. GT eventually recognizes stars as a special case of connected graphs, discovering trees along the way. A second motivation for concept generalization is the conjecture and proof of relations among properties. When a property is highly-detailed, the entailed matching can be expensive. Reasoning about a more general case, which typically has a simpler form, may be much more efficient. For example, if A is a special case of B and B is disjoint from C, A will also be disjoint from C. Often GT's discovery and proof that B is disjoint from C is faster.

4.3 New Concepts Discovered by Merger

As noted in Section 3.2, GT has several techniques to construct the merger of two properties. The merger of property p_1 with property p_2 represents the intersection of P_1 with P_2 and is readily computable in most instances. GT has discovered trees, for example, by constructing the merger of ACYCLIC and CONNECTED.

5. Discovery Modes and Search

Up to this point, GT has been described as a system that represents graph theory concepts as frames and has the ability to generate examples, conjecture and prove theorems, and construct new concepts. This section details the spectrum of GT's operational modes, from those where GT takes the least initiative to those where GT appears totally self-directed.

In the first of its operational modes, GT simply performs under interactive guidance: the user specifies a database of initial concepts and example graphs, and then directs GT to a specific task, called a *project*. Possible projects are

- generate some examples of property p
- test to see if property p_1 subsumes property p_2
- test to see if property p_1 is equivalent to property p_2
- construct the merger of property p_1 with property p_2
- generalize property p
- specialize property p

Each project is performed according to the algorithms described above. The non-determinism discussed in Section 2 may produce different traces or even different outcomes for different executions of the same project (e.g., different examples may be generated), but the theorem proving algorithms always produce consistent results.

In its first mode, GT follows external directives to formulate and execute projects, ones presumably based on the user's expectation that such investigations will result in additional mathematical knowledge. In its second mode, GT formulates its own projects and places them on its agenda. In this mode the user specifies the kind of project GT is to suggest:

- example generation
- subsumption
- equivalence
- merger
- generalization

- specialization

Clearly the set of GT-originated project types is identical to those posable by the user in the first mode. The user may also indicate a *focus* for these formulated projects, i.e., a property that is considered particularly interesting. If a focus is designated, then each conjecture will involve it. In its second mode, once GT formulates the projects, the user designates tasks to be executed.

Given a knowledge base of k properties, there are potentially $2k^2+k$ projects on the first pass, i.e., before newly-created properties participate in project formulation. How does GT limit search through such a space? The human mathematician has two primary sources of evidence on which to base project formulation: definitions and examples. Unlike AM whose cumbersome LISP code limited it to conjecture based solely upon examples, GT is capable of reasoning both from p-generator definitions and from specific graphs, either seeds or generated examples. (The latter is an example of what Thagard and Holyoak (Thagard and Holyoak 1985) call "instance-based generalization.") These two sources support the formulation of projects in a variety of ways.

When presented with a definition for a concept, most mathematicians immediately construct examples. Thus GT recommends a project to generate examples of any property for which it lists "too few" (as determined by a global variable) examples.

A mathematician presented with non-empty classes P and Q from a universe U is trained to explore potential relations between the classes by examining whether or not each of the labeled regions in Figure 5 is empty. GT models this strategy with conjectures about subsumption and merger. The standard mathematical questions, and their GT equivalents, are

- Is p a subset of Q? GT explores this by a conjecture that q subsumes p.
- Is P a superset of Q? GT explores this by a conjecture that p subsumes q.
- Is P equal (equivalent) to Q? GT explores this by two conjectures, that p subsumes q and that q subsumes p.
- Are P and Q disjoint (mutually exclusive)? GT explores this by a suggestion to merge p and q.

Thus the theorems that GT conjectures are statements about set-theoretic relations between

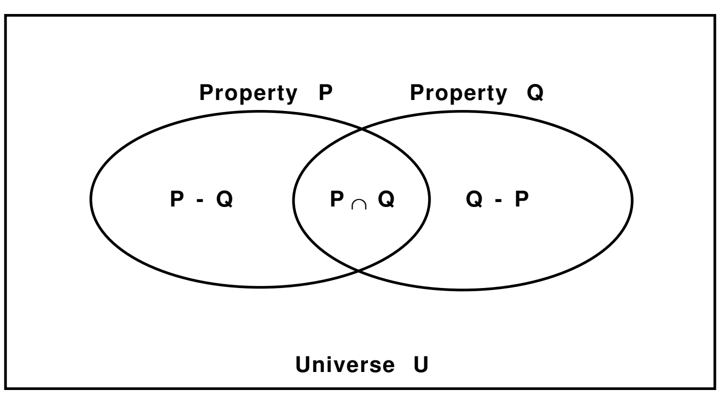

Figure 5: Potential Set-Theoretic Relations between Two Classes

classes of graphs. The analysis in Figure 5 may, of course, be extended to more than two sets. For example, either set may be replaced by the set $R \cap S$. Since GT can use merger to construct $R \cap S$, GT conjectures encompass all four of the theorem types at the beginning of Section 3.

The first two mathematical questions, on subset/superset conjectures, are based upon both seeds and definitions. If a seed for p is already on record in GT as a seed for some other property q, a subset/superset relation may exist between P and Q. GT examines such (p,q) pairs for additional supporting evidence from the property definitions. In particular, GT looks for:

- a degree of similarity in the seed sets for p and q (in decreasing order of significance: equal sets, one a subset of the other, a non-null intersection)
- seeds of property p that are known to have the property q
- a degree of similarity between the operators for p and q (i.e., which primitives are employed and in what groupings)

The strong focus on seeds is justified both by their role as prototypes and by efficiency; seeds tend to be small and few in number.

Before GT's heuristics explore the third mathematical question, the equivalence of p and q, they require that the two associated subsumptions have been either proved or conjectured. Alternative definitions (*characterizations*) of classes are common in mathematics because they support conjecture and, therefore, research. GT demonstrates such use of alternative definitions. Consider, for example, the class of graphs known as *chains*, some examples of which appear in Figure 6. GT has two different definitions of chain:

$$\mathrm{CHAIN}_1 = \langle A_{xv}A_{vy}D_{xy}A_v, \{K_2\}, [x,y \in V, v \notin V, xy \in E]\rangle$$

and

$$\mathrm{CHAIN}_2 = \langle A_{xy}A_y, \{K_2\}, [x \in V, y \notin V, d(x) = 1]\rangle$$

GT is also given definitions of TREE and CYCLE:

$$\mathrm{TREE} = \langle A_{xy}A_y, \{K_1\}, [x \in V, y \notin V]\rangle$$

$$\mathrm{CYCLE} = \langle A_{xv}A_{vy}D_{xy}A_v, \{K_3\}, [x,y \in V, v \notin V, xy \in E]\rangle$$

Based on the operators, CHAIN_1 suggests that a chain may be a cycle, and CHAIN_2 suggests that a chain may be a tree. GT formulates and investigates both conjectures, and discovers that the first is incorrect and the second correct.

The fourth mathematical question, a conjecture about disjointness, is really a conjecture that a merger will fail. Thus a conjecture in GT about the disjointness of p and q is expressed as a plan to merge p and q. If the seed sets for p and q are disjoint, the possibility of the disjointness of P

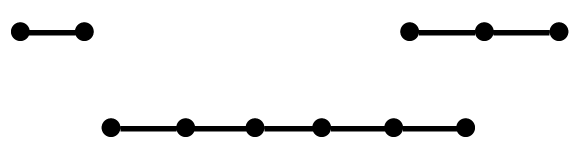

Figure 6: Some Examples of Chains

and Q will be conjectured in the form of a plan to attempt the merger of p and q.

Projects to generalize and specialize are motivated either by the fact that the concept is a focus or by a dearth of projects on the agenda. In such circumstances GT formulates a project to generalize or specialize a property if it has never done so.

In its second mode, GT makes its own suggestions for appropriate projects and leaves the choice to the user. In its third mode, GT not only suggests projects for its agenda, but also selects and executes them. Unless some difficulty arises (e.g., an unsolvable system of equations during merger or an unsuccessful search for a common seed), there is no interaction with the user. Projects are ranked (based on the kind of supporting evidence and any focus specified by the user), sorted in the agenda, and executed in order. The third mode is surprisingly well-controlled because the kinds of conjectures made are dependent upon the knowledge base itself. For example, the priority is always to construct links between existing properties. Only when those possibilities have been exhausted will GT turn to inventing new ones. This determined effort to construct a net of properties results in the early identification of equivalent properties, and helps to control the combinatoric explosion. The third mode can perform as a model of knowledge acquisition. If GT is given a pre-existing knowledge base of concepts and one additional concept as a focus, GT will generalize and/or specialize the focus concept until it is able to link it into the knowledge base. Thus knowledge acquisition consists of the construction of intermediate concepts linked to both the focus concept and the knowledge base.

6. Results and Significance

According to Michalski's characterization of learning systems (Michalski 1986), GT learns both by observation (of its input examples and definitions) and by discovery (upon construction of new examples and properties). GT expands its knowledge about a concept by generating examples of it and by determining its relation to other concepts. GT inductively infers conjectures from examples and definitions, and also proves deductively from the same definitions. GT formulates examples of known concepts and also defines and explores new concepts.

GT transforms its representation about a concept in the knowledge base. Figure 7 displays the ACYCLIC frame both before and after one of GT's runs. No specific tasks were input, only the general directive to explore the knowledge base. GT formulates its own conjectures and then attempts to construct proofs for them based on the structure of the definitions. The modifications to the representation for ACYCLIC constitute learning as defined in (Michalski 1986). Clearly GT learns how ACYCLIC relates to other concepts, and constructs and stores additional examples of acyclic graphs. GT learns about graph theory by conjecturing and exploring simple relations among graph properties.

GT is able to conjecture theorems in graph theory. Conjecture is driven by extremal examples and definitions. Example-driven discovery is based upon prototypical graphs (seeds) that are extremal cases of individual properties and therefore likely to be rich in associations. Definition-driven discovery focuses upon the transformations that change one graph with a property into another graph with the same property. The requirement that a definition be complete effectively limits such transformations to minimal changes. For example, a connected graph may be transformed by adding a new vertex with one edge to an old vertex. Requiring that the new vertex be connected to more than one old vertex would create a different, more restricted, set of graphs. The minimality of these changes and the limited vocabulary of operator primitives makes relations between the transformations in the definitions more readily apparent.

GT is able to prove theorems in graph theory that it has conjectured. Proofs rely heavily on a procedure to test for subsumption and a procedure for merger to represent graphs with more than one property. Running on a Symbolics 3675 in Symbolics Common Lisp, GT successfully conjectures and proves, among other theorems, the following:

- Every tree is acyclic.

	Initial Formulation	**After Execution**
Property name	ACYCLIC	ACYCLIC
Number-of-seeds	1	1
Seed-set	$\{K_1\}$	$\{K_1\}$
Function	$A_x + A_{yz}A_z$	$A_x + A_{yz}A_z$
Sigma	$y \in V, x,z \notin V$	$y \in V, x,z \notin V$
Origin	input	input
Example-list	(K_1)	(ACYCLIC-3, ACYCLIC-2, K_1)
Extremal-cases	(K_1)	(K_1)
Delta-pairs	((1 0)(1 1))	((1 0)(1 1))
Subsumes	nil	(ACYCLIC-MERGED-WITH-CONNECTED CHAIN TREE)
Cannot-be-shown-to-subsume	nil	(CONNECTED EQUIV-CONNECTED)
Subsumed-by	nil	(IS-A-GRAPH)
Cannot-be-shown-subsumed-by	nil	(CONNECTED EQUIV-CONNECTED CHAIN TREE ACYCLIC-MERGED-WITH-CONNECTED)
Is-equivalent-to	nil	nil
Cannot-be-shown-equivalent-to	nil	(CHAIN TREE CONNECTED IS-A-GRAPH EQUIV-CONNECTED)
Merger-created-with	nil	(CONNECTED)
Merger-explored-with	nil	(CHAIN TREE)
Has-been-generalized	nil	nil
Has-been-specialized	nil	T
Variables-have-been-identified	nil	T
Iteration-has-been-forced	nil	T

Figure 7: What GT Learns about ACYCLIC

- Every tree is connected.
- The set of acyclic, connected graphs is precisely the set of trees.
- There are no odd-regular graphs on an odd number of vertices.

GT discovers new mathematical concepts by syntactic changes whose semantics are well-understood and accessible to the program. The key in GT is a more transparent and flexible class definition, one that generates guaranteed examples, constructs efficient intersections, and creates from a broad descriptive vocabulary. These concepts form a rich knowledge base conducive to further mathematical discovery. From the examples and definitions of Section 4, the following is evident:

> **THEOREM**: The heuristics used by GT to constrain/relax any definition of a graph property P construct valid specializations/generalizations of P.

This theorem guarantees that the definitions GT constructs are, in fact, graph properties. It also justifies the hierarchical links GT inserts during the discovery process.

Since GT's knowledge base may be initialized as any set of graph properties, a concept may be discovered in more than one way. In one experiment, GT begins only with the definition of a graph and the heuristics described here, and discovers, among other concepts:

- acyclic graphs
- connected graphs
- bipartite graphs
- trees
- stars

GT is able to incorporate all of these correctly into its hierarchical knowledge structure. A demonstration during which all of these discoveries take place requires approximately three and a half minutes of elapsed time. In another experiment, GT begins with a small initial knowledge base of concept definitions, links them together and then generalizes the focus concept "star" until it is able to link it into its knowledge base. During the elapsed time (less than one minute) required to do this, GT also discovers "tree."

When GT "invents" a new property definition, it is subjected to careful scrutiny before a concept frame is created for it. Many generated definitions are trivial, i.e., they may iterate only once or twice, or even be limited entirely to their seed set. Other definitions, intended as a specialization of some parent concept, may very quickly produce many more examples than were known for the parent. Still other definitions, intended as generalizations of some parent concept, may produce only graphs already known as examples of the parent. All of these constructs are deemed uninteresting and rejected as potential concepts.

Some of GT's discovery paths are a bit surprising. For example, although TREE is a special case of CONNECTED, definitions for both concepts appear during a single exploration cycle. In another unanticipated action, when STAR is being generalized, GT moves backwards, first to TREE and then to a definition for "connected graphs with loops," skipping over ACYCLIC and CONNECTED completely. Even well-planned inductive leaps do not always arrive where expected.

Exhaustive search, rather than a burden, seems to be one of GT's strengths. The richness of the semantic network it constructs is due to its extensive exploration. In the META-DENDRAL tradition, GT can afford exhaustive search because its representation is highly controlled. The property definitions support reasoning without recourse to empirical data. GT's design, however, does not advocate the abandonment of inductive inference from empirical data in the modeling of a scientific research domain. Indeed, GT "doodles" graphs as a spur to inquiry. Rather, GT's design suggests that abstract reasoning about a class from a correct, complete definition is a powerful complement to inductive inference.

What is the scientific significance of this particular domain-specific discovery system? First, GT integrates a variety of artificial intelligence techniques to provide greater power. Unlike its machine learning predecessors, GT combines both examples and theory to drive discovery. Unlike theorem provers, GT produces its own conjectures. Second, GT formulates a new and general approach to the representation of mathematical information, one that is not limited to graph theory. GT's approach to mathematical data enables the encoding, organization, and manipulation of the heterogeneous knowledge typical of mathematical texts. In particular, the recursive $<f,S,\sigma>$ representation shows how an infinite class of mathematical objects can be described as a *property* to support reasoning about them as a collection. The same representation imposes a uniformity that enables reasoning about *theorems*, the relations among concepts, and enables the description of results as *concepts*, collections of information. Finally, GT shows how a semantic network of those concepts can be an effective organization of information to support proof construction. None of this approach is philosophically limited to graph theory; it should be extensible to other mathematical domains, and perhaps even other scientific domains.

Although its example-generation technique could be extremely useful, GT is not intended as a practical tool for researchers in graph theory. Instead, GT is intended to provide insight into the mathematical research process itself. When its exhaustive search strategy eventually weakens in the face of more demanding tasks, GT can be enhanced to model behaviors that people use to support research. Two examples of this potential for growth are isomorphism and counterexamples. The recognition of two objects, in particular, two graphs, as "fundamentally equivalent" is non-trivial, yet essential to control the size of GT's knowledge base. A better recognition algorithm for isomorphic graphs is under development. Counterexamples are extremely significant to mathematicians; GT is currently being modified to retain them and take advantage of the information they provide.

7. Future Work

Lenat's work with AM convinced him that, as the research area within mathematics changed (from, say, set theory to number theory) new discovery heuristics were required (Lenat 1983). GT is designed to work within a single area of mathematics; no need for new heuristics is anticipated. Instead, plans for GT's future development are based upon the power and flexibility of the p-generator representation.

Michalski's and Dietterich's work (Dietterich and Michalski 1983; Michalski 1983) on generalization rules for concept acquisition provide some excellent suggestions for concept discovery in GT. GT already embodies both selective and constructive generalization techniques, such as the "dropping condition" rule (as selector relaxation) and the "closing interval"" rule (as a merger heuristic). Other rules currently under consideration and/or development include extending reference, counting arguments, and internal disjunction. GT's descriptive ability lies in the number and nature of the primitive operators permitted in f and of the selector descriptions permitted in σ. As the set of such operators and descriptions is extended, a lattice of descriptive languages (detailed in (Epstein 1983)) can be constructed. Such an "extended" language offers additional alternatives, and ordinarily has greater expressive power (as measured by the number of graph properties it defines) than GT's current representation. In turn, operations with an extended language are likely to require more computer resources. Within the discovery framework described here, plans exist to extend the p-generator language for the representation of directed graphs and, eventually, for labeled graphs. These extensions will also provide a test-bed for the study of performance under representational shifts.

The key to the most interesting specializations, those involving additional descriptions in σ, is the language in which those descriptions may be written. Utgoff (Utgoff 1986) warns that, unless the [σ-]language is extensible, GT may not be able to access many interesting ideas. Ways to have GT extend the σ-language itself are currently being studied. Despite substantial empirical support, the existence of a definition of the form $<f,S,\sigma>$ for every property p remains an open question.

At the moment, GT has a variety of small knowledge bases of input concepts and search has been exhaustive within the list of generated conjectures. GT's discovery is thus primarily theory-driven, based as it is on definitional structure. As the input knowledge base increases, however, and as GT becomes more proficient at discovering interesting concepts of its own, control during search will become an issue. GT now numerically rates the projects on its agenda based on supporting evidence, in a fashion similar to AM, although it does not assign "worths" to individual concepts. GT has a threshold for rating values, and only executes the projects that meet it. In anticipation of a combinatoric explosion, work is under way to use additional example-based reasoning, particularly counterexamples, to evaluate the agenda and guide search. Thus discovery will derive additional data-driven support, while maintaining its theory-driven component. GT's knowledge base will be expanded with more concepts gleaned from the benchmark texts. Mathematicians studying interesting sets of graph properties are invited to submit them to GT. The shell of GT is a domain-independent research tool for recursive property description. Work is underway to apply this shell to mathematical domains other than graph theory. Finally, extensions to this shell are currently under development to model a variety of other research behaviors.

Acknowledgments

This research was supported in part by NSF grant #DCR-8514395. The author thanks N.S. Sridharan for his thoughtful discussions and comments, Virginia Teller for her careful attention to clarity and detail, and an anonymous referee for very useful comments on an earlier draft of this paper.

Bibliography

Bondy, J. and Murty, U. 1976. *Graph Theory with Applications* . New York: North-Holland.

Buchanan, B. G. and Mitchell, T. M. 1978. Model-Directed Learning of Production Rules. In D. A. Waterman, & F. Hayes-Roth (Ed.), *Pattern-Directed Inference Systems,* 297-312. New York: Academic Press.

Carbonell, J. G., Michalski, R. S. and Mitchell, T. M. 1983. An Overview of Machine Learning. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Ed.), *Machine Learning: An Artificial Intelligence Approach,* 3-23. Palo Alto: Tioga Publishing.

Dietterich, T. G. and Michalski, R. S. 1983. A Comparative Review of Selected Methods for Learning from Examples. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Ed.), *Machine Learning: An Artificial Intelligence Approach,* 41-81. Palo Alto: Tioga Publishing.

Emde, W., Habel, C. U. and Rollinger, C.-R. 1983. The Discovery of the Equator or Concept Driven Learning. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, 455-458. Karlsruhe: William Kaufmann, Inc.

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

Epstein, S. L. 1987. Languages for Problem Solving in Graph Theory. In J. C. Boudreaux, B. W. Hamill, & R. N. Jernigan (Ed.), *The Role of Language in Problem Solving 2,* 261-300. New York: North-Holland.

Harary, F. 1972. *Graph Theory* . Reading: Addison-Wesley.

Laird, P. G. 1986. Inductive Inference by Refinement. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, 472-476. Philadelphia, PA.

Langley, P., Bradshaw, G. L. and Simon, H. A. 1983. Rediscovering Chemistry with the BACON System. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Ed.), *Machine Learning: An Artificial Intelligence Approach,* 307-329. Palo Alto: Tioga Publishing.

Langley, P., Zytkow, J. M., Simon, H. A. and Bradshaw, G. L. 1986. The Search for Regularity: Four Aspects of Scientific Discovery. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Ed.), *Machine Learning: An Artificial Intelligence Approach,* 425-469. Palo Alto: Tioga Publishing.

Lee, W. D. and Ray, S. R. 1986. Rule Refinement Using the Probabilistic Generator. In *Proceedings of the Fifth National Joint Conference on Artificial Intelligence*, 442-447. Philadelphia, PA:

Lenat, D. B. 1976. *AM: An Artificial Intelligence Approach to Discovery in Mathematics*. Ph.D. thesis, Department of Computer Science, Stanford University.

Lenat, D. B. 1983. The Role of Heuristics in Learning by Discovery: Three Case Studies. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Ed.), *Machine Learning: An Artificial Intelligence Approach,* 243-306. Palo Alto: Tioga Publishing.

Lenat, D. B. and Brown, J. S. 1984. Why AM and EURISKO Appear to Work. *Artificial Intelligence*, 23 (3): 249-268.

Michalski, R. S. 1983. A Theory and Methodology of Inductive Learning. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Ed.), *Machine Learning: An Artificial Intelligence Approach,* 83-134. Palo Alto: Tioga Publishing.

Michalski, R. S. 1986. Understanding the Nature of Learning: Issues and Research Directions. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Ed.), *Machine Learning: An Artificial Intelligence Approach,* 3-25. Palo Alto: Tioga Publishing.

Michalski, R. S. and Stepp, R. E. 1981. Concept-based Clustering versus Numerical Taxonomy, 1073, Department of Computer Science, University of Illinois.

Michalski, R. S. and Stepp, R. E. 1983. Learning from Observation: Conceptual Clustering. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Ed.), *Machine Learning: An Artificial Intelligence Approach,* 331-363. Palo Alto: Tioga Publishing.

Michener, E. R. 1978. Understanding Understanding Mathematics, AI MEMO-488, LOGO MEMO-50, MIT.

Nordhausen, B. 1986. Conceptual Clustering Using Relational Information. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, 508-512. Philadelphia.

Ore, O. 1962. *Theory of Graphs* . Providence: American Mathematical Society.

Ritchie, G. D. and Hanna, F. K. 1984. AM: A Case Study in AI Methodology. *Artificial*

Intelligence, 23 (3): 269-294.

Shapiro, E. Y. 1981. An Algorithm that Infers Theories from Facts. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, 446-451. Vancouver, B.C.

Stepp, R. E. and Michalski, R. S. 1986. Conceptual Clustering: Inventing Goal-Oriented Classifications of Structured Objects. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Ed.), *Machine Learning: An Artificial Intelligence Approach - Volume II,* 471-498. Palo Alto: Tioga Publishing.

Thagard, P. and Holyoak, K. 1985. Discovering the Wave Theory of Sound: Inductive Inference in the Context of Problem Solving. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, 610-612. Los Angeles: Morgan Kaufmann.

Utgoff, P. E. 1986. Shift of Bias for Inductive Concept Learning. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Ed.), *Machine Learning: An Artificial Intelligence Approach - Volume II,* 107-148. Palo Alto: Tioga Publishing.