This paper appeared in 1991 in *Applied Intelligence*, 1 (1): 7-33.

Knowledge Representation for Mathematical Discovery
Three Experiments in Graph Theory

Susan L. Epstein and N.S. Sridharan

Affiliation of Authors

Epstein: Department of Computer Science, Hunter College of the City University of New York, 695 Park Avenue, New York, NY 10021.

Sridharan: Corporate Technology Center, FMC Corp., 1205 Coleman Avenue, Santa Clara, CA 95052

Financial Support

This research was supported in part by NSF grant #DCR-8514395.

Abstract

This paper describes the nature of mathematical discovery (including concept definition and exploration, example generation, and theorem conjecture and proof), and considers how such an intelligent process can be simulated by a machine. Although the material is drawn primarily from graph theory, the results are immediately relevant to research in mathematical discovery and learning.

The *thought experiment*, a protocol paradigm for the empirical study of mathematical discovery, highlights behavioral objectives for machine simulation. This thought experiment provides an insightful account of the discovery process, motivates a framework for describing mathematical knowledge in terms of object classes, and is a rich source of advice on the design of a system to perform discovery in graph theory. The evaluation criteria for a discovery system, it is argued, must include both a set of behavior to display (behavioral objectives) and a target set of facts to be discovered (factual knowledge).

Cues from the thought experiment are used to formulate two hierarchies of representational languages for graph theory. The first hierarchy is based on the superficial terminology and knowledge of the thought experiment. Generated by formal grammars with set-theoretic semantics, this eminently reasonable approach ultimately fails to meet the factual knowledge criteria. The second hierarchy uses declarative expressions, each of which has a semantic interpretation as a stylized, recursive algorithm that defines a class by generating it correctly and completely. A simple version of one such representation is validated by a successful, implemented system called Graph Theorist (GT) for mathematical research in graph theory. GT generates correct examples, defines and explores new graph theory properties, and conjectures and proves theorems.

Several themes run through this paper. The first is the dual goals, behavioral objectives and factual knowledge to be discovered, and the multiplicity of their demands on a discovery system. The second theme is the central role of object classes to knowledge representation. The third is the increased power and flexibility of a constructive (*generator*) definition over the more traditional predicate (*tester*) definition. The final theme is the importance of examples and recursion in mathematical knowledge. The results provide important guidance for further research in the simulation of mathematical discovery.

Table of Contents

Knowledge Representation for Mathematical Discovery: Three Experiments in Graph Theory

As an ill-structured problem, discovery is potentially the most difficult subfield of machine learning. Successful discovery requires the representation and manipulation of quantities of information, the efficient storage and retrieval of that information, and a processing strategy that focuses upon relevant segments of that information and pursues it with resources appropriate to the information's significance. This paper explores the origin and nature of knowledge representation to support the efficient simulation of mathematical research.

A system, be it human or machine, that performs discovery in mathematics is confronted with a rich and varied body of knowledge. There are definitions, examples, theorems, algorithms, conjectures, and proofs, all linked together in an intricate structure. Search through so rich a structure, however, is likely to be combinatorially explosive. A successful discovery system must therefore represent information in a way that minimizes irrelevancies and intuits fruitful avenues of exploration during search.

The mathematical discovery systems of Epstein, Gelernter, and Lenat, each succeed because their representations curtail search. The Graph Theorist, GT [8], uses a set of powerful representation languages for object classes to support mathematical research in graph theory. GT generates examples, defines and explores new concepts, and conjectures and proves theorems. Gelernter's Geometry Theorem Proving Machine [10] uses a diagram of the most general case of the hypotheses to guide its search for proofs of theorems in rectilinear plane geometry. Lenat's AM [14] uses generated examples of mathematical concepts beginning with set theory and reasons inductively from frame representations for them.

There are certain basic expectations for knowledge representation in a discovery system: the representation must express the information (*factual knowledge)* people discover, and it must do so in a way which is flexible enough to imitate the broad spectrum of knowledge processes (*behavioral objectives)* observed in people as they do discovery. These are very different kinds of criteria. Behavioral objectives are known to the system designer in advance, and only an implementation can demonstrate their achievement. On the other hand, factual knowledge to be discovered is not fully disclosed in advance, and yet the discovery system's knowledge representation must be theoretically able to express it.

This paper proposes, examines, and evaluates three experiments in knowledge representation and mathematical discovery. Although the experiments are all in the subdomain of graph theory, their results are relevant to the more general problems of mathematical discovery. The first is a *thought experiment*, an extended example, constructed (abstracted) from human protocols of discovery sessions in graph theory. Many AI systems have behavioral objectives defined by introspection, principally that of their authors. Several systems, intended to provide cognitive modeling, use formal methods of protocol collection. We have combined aspects of both in our paradigm of thought experiments. We had several subjects attempt discovery and record the trace of their thinking, and took as the objective for our system a composite of the several phenomena observed in that trace. Analysis and interpretation of these protocols compiles a broad range of mathematical behavioral objectives to replicate, and a variety of clues on how knowledge can be represented to make those behaviors accessible to computation. The second and third experiments use those clues to formulate and then evaluate hierarchies of representational languages designed to support discovery in graph theory. The last two experiments are held to the discovery criteria described above: behavioral objectives and factual knowledge.

The thought experiment serves to direct our inquiry about discovery systems in graph theory. It highlights the declarative and procedural knowledge required, and motivates our subsequent development of mathematics as the study of large object classes and relations among them. Specific examples and constructional derivation are also important in the discovery process. In our quest for representations that incorporate all of these features, the second and third experiments were run. The second experiment is a reasonable supposition, with a traditional AI approach to definitions as testing algorithms. It fails to meet the criterion of factual knowledge, because it is unable to express all the properties of interest. The third experiment is a more elaborate recursive approach with definitions that are generating algorithms based on examples. Its implementation, GT, has succeeded in modeling much of the mathematical discovery we saw in the paradigm.

A brief guide through the paper follows. The thought experiment appears, as a trace with commentary and discussion, in the first section. The second section, based on observations from the first, formulates an original approach to mathematical knowledge that focuses on classes of objects. The third section discusses some issues in the relations between a knowledge representation language and its domain. The second experiment is described in Sections 4 and 5. Section 4 begins with the definition of the target set of factual knowledge for research in graph theory. Because the thought experiment repeatedly references edges, Section 4 constructs a formal (*edge-set*) language to support discovery in graph theory. This language describes graphs by set-theoretic relations on their edge sets. To address the important problems of expressive power and extensibility, a hierarchy of representation languages is useful. From the language of Section 4, Section 5 constructs a hierarchy of edge-set languages and evaluates them. The third experiment, in Section 6, takes additional cues from the thought experiment to construct a second hierarchy of languages (the *R-languages*) to support discovery in graph theory. The success of one R-language, used in the implementation of GT, is summarized in Section 6. Finally, Section 7 recapitulates the paper's results, evaluates them against the discovery criteria, compares related work, and extracts lessons to be learned for the larger domain of mathematical discovery.

1. A Thought Experiment

Since the mathematical discovery process is not only ill-structured, but also ill-defined, this section provides an extended example of the intelligent behavior(s) to be simulated by a mechanistic mathematician. This *thought experiment* poses a topic for exploration to a student, and describes the results in detail. As a recorded instance of mathematical discovery, it also proves to be a rich source of ideas about how to achieve those goals.

The discoverer in the thought experiment is a hypothetical student mathematician, one familiar with mathematical notation and proof procedures, and acquainted with rudimentary set theory and certain simple properties of graphs. The topic of exploration is a relatively simple, well-defined property in graph theory, one totally unfamiliar to the student. The trace [25] of the student investigating the topic is actually a composite of several students' (S. Epstein, E. Schnabel, and N.S. Sridharan) exploratory sessions, abstracted and somewhat polished for coherence. The trace itself is divided into three episodes; they appear in Figures 2, 3, and 4. The key definitions, intended to provide focus and support for the student (and perhaps refresh the memory of the reader) appear with the problem in Figure 1.

As you read the trace, and the interpretation of it that follows, consider the following questions:

- What conditions might be necessary for an AI system to discover mathematical knowledge?
- What role do examples play?
- What kind of tasks arise?
- What heuristics appear to guide discovery?

A *graph G* is an ordered pair $<V,E>$, where V is an arbitrary, non-empty finite set and E is any subset of the Cartesian product $V \times V$. The elements of V are called the *vertices* of G, and the elements of E are called the *edges* of G. If $x \in V$ but there is no edge (x, y) or (y,x) in E, x is said to be *isolated*. The *reverse* of an edge (x,y) is defined to be the edge (y,x).

A graph $G = <V,E>$ is *undirected* when an edge is in E if and only its reverse is in E; otherwise the graph is *directed*. In an undirected graph, the *degree* of a vertex v is $d(v) = |\{(x,y) \mid x \neq y, x = v\}|$, the number of other vertices with which it shares an edge. Let $|V| = n$, $|E| = m$, $I_n = \{1,2,\ldots,n\}$, and $A_n = \{(x,y) \mid x \neq y, x,y \in I_n\}$.

The *empty graph* O_n is $<I_n,\emptyset>$. The empty graph has n vertices and no edges.

The *complete graph* K_n is $<I_n,A_n>$. The complete graph has n vertices and all possible unordered edges between distinct vertices. $max(|E|) = |A_n| = n(n-1)/2$.

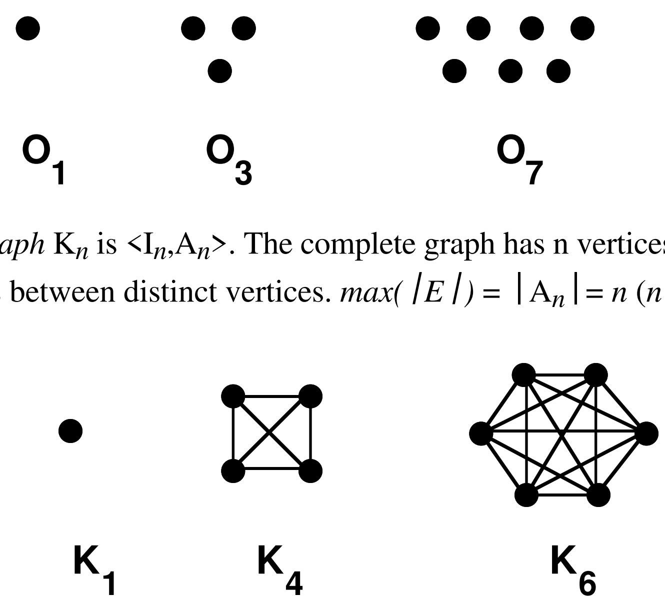
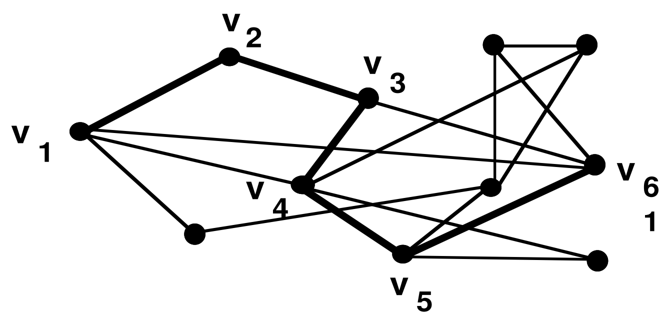


Figure 1: The Problem and its Background (Part 1)

A *path* $x, xv_1, v_1, v_1v_2, \ldots, v_k, v_ky$ (abbreviated as $\langle x, v_1, \ldots, v_k, y \rangle$ or simply $\langle x, y \rangle$) in a graph $G = \langle V, E \rangle$ is an alternating sequence of distinct vertices in V and edges in E that begins with x and ends with y. A graph $G = \langle V, E \rangle$ is *connected* if and only if for every distinct pair of vertices x and y there exists a path between them; otherwise a graph is *disconnected*.

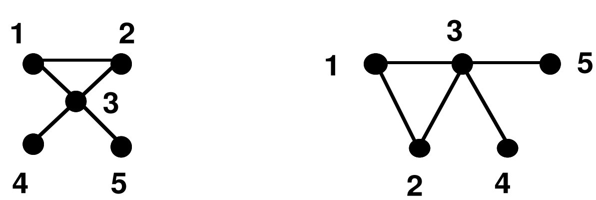


A path from v_1 to v_6

Two graphs $G_1 = \langle V_1, E_1 \rangle$ and $G_2 = \langle V_2, E_2 \rangle$ are *isomorphic* ($G_1 \sim G_2$) if and only if there exists a one-to-one, onto mapping f from V_1 to V_2 such that $(f(x), f(y)) \in E_2$ if and only if $(x,y) \in E_1$. Isomorphism may be used to place the set of all graphs on n vertices into equivalence classes. Then U_n is the set of all graphs on n vertices that are distinct up to isomorphism and the set of all graphs is

$$U = \bigcup_{n=0}^{\infty} U_n$$

For convenience, each member of U will be represented by a single graph and referred to as such. Isomorphism preserves a number of topological properties of graphs, including number of connected components, degree sequence, and planarity.



A pair of graphs with vertices labeled to show isomorphism

Each graph $G = \langle V, E \rangle$ has associated with it a *complement,* a graph $G^C = \langle V, E^C \rangle$ on the same vertices, whose edge set E^C is the complement of E with respect to all possible edges between distinct vertices in V, i.e., for $|V| = n$, $E^C = E_n - E$.

The Problem: A graph G is said to be *self-complementary* if and only if $G \sim G^C$. Investigate the existence and characteristics of undirected self-complementary graphs.

Figure 1: The Problem and its Background (Part 2)

1.1 Episode 1: Example Generation

This section is to be read in conjunction with Figure 2. To discover interesting and useful properties of self-complementary graphs it is necessary first to find some. The first episode is therefore a search for examples of the property.

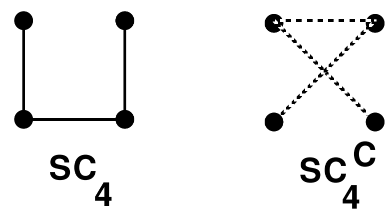
Mathematicians know that extreme cases are often more efficiently represented and manipulated than randomly generated objects. An extreme is measured in terms of some metric; the most obvious metric on a graph is n, the number of its vertices. Since a finite graph may have any finite number of vertices, the only extreme case is $n = 1$. Thus the search begins with the only graph on one vertex, and Discovery 1 comes at the examination of the very first graph.

As a mathematician, the student presumably has a predilection for induction, which begins with an extreme case and reasons about its relation to subsequent examples. Hence, additional examples are necessary. Discovery 2 is the result of exhaustive search, i.e., both of the non-isomorphic graphs on two vertices and the 4 graphs on three vertices were tested. Despite this failure, exploration continues, because one example has been found and the searcher's resources are not yet exhausted.

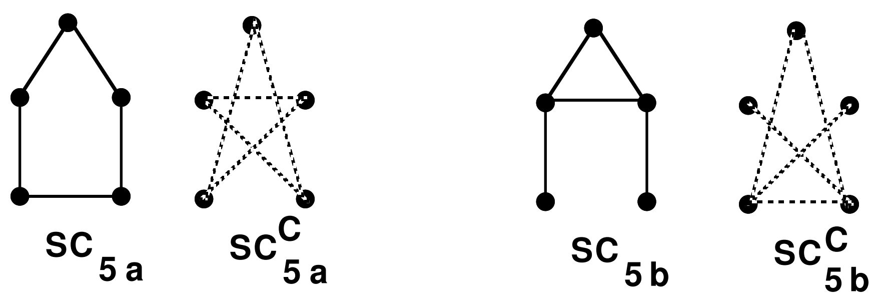
Discovery 3, of a second self-complementary graph, reinforces the impetus to seek additional examples. (By this point it may be difficult to generate the graphs methodically; [11] recommends Polya's formula and Riordan's algorithms.) After Discovery 4 there is more than one self-complementary graph, but finding the second required examining 11 additional graphs, many more than finding the first did. The student wonders how rare self-complementary graphs are, and whether there is ever more than one for the same n value. Discoveries 5 and 6, the existence of two self-complementary graphs among the 34 on five vertices, resolve these questions. During the first episode, four self-complementary graphs have been discovered among 52 distinct graphs. The rejected 48 graphs form a broad set of examples of "not-self-complementary."

Rather than continue to generate more examples, the student turns to explain the relative sparsity of the example graphs. In the next two episodes, the discovery mechanism makes observations and inductive generalizations, forms conjectures based on examples, and verifies some of those conjectures.

Heuristic: Begin the investigation by constructing examples (to show the existence of self-complementary graphs). **Heuristic:** One place to start is to try extreme cases.
Discovery 1: O_1, the empty graph on one vertex, is self-complementary.
Heuristic: Try the next higher cases.
Discovery 2: No graph on 2 or 3 vertices is self-complementary.
Discovery 3: One graph SC_4 with n = 4 is self-complementary.



Discovery 4: The remaining graphs with n = 4 are not self-complementary.
Discovery 5: Two graphs SC_{5a} and SC_{5b} with n = 5 are self-complementary.



Discovery 6: The remaining graphs with n = 5 are not self-complementary.

Figure 2: Episode 1 - Example Generation

1.2 Episode 2: Explaining Non-Examples

This section is to be read in conjunction with Figure 3. Ostensibly, the second episode seeks general criteria for the existence of self-complementary graphs by exploring the nature of graphs that are not self-complementary, but underlying the episode is interest in the construction of self-complementary graphs.

The definition of self-complementary in Figure 1 is a *tester*, a predicate that determines whether an input example meets or fails its criteria. A *generator* for a set is an input-free automaton that, once started, produces the set. A generator for a set is said to be *correct* if and only if every graph output by the generator is in the set. A generator for a set is said to be *complete* if and only if for each object in the set there exists a finite sequence of iterations of the generator whose final output is that object. A generator defines a set if and only if it is both correct and complete with respect to

Heuristic: Explain Discovery 2, i.e., why there are no self-complementary graphs on 2 or 3 vertices.
Heuristic: Seek explanation in terms of n.
Fact: $max(|E|) = n(n-1)/2$
Fact: $max(|E|) = 1$ for $n = 2$
Fact: $max(|E|) = 3$ for $n = 3$
Explanation: If G is self-complementary, $|E| = |E^C|$ because $G \sim G^C$ and $|E| + |E^C| = max(|E|)$ by definition of "complement." Thus $max(|E|)$ must be even.
Discovery 7: Generalization of Discovery 2: No graph on any number of vertices is self-complementary if $max(|E|)$ is odd.
Heuristic: Assimilate this to assist in the search for new examples.
Discovery 8: Since $|E| + |E^C| = max(|E|)$ and $max(|E|) = n(n-1)/2$,
$|E| = n(n-1)/4$.
Discovery 9: For G to be self-complementary, n must be congruent to 0 or to 1, modulo 4. Thus

$$m = \frac{n\,(n\,\text{-}1)}{2} \ \text{ and } \ n \bmod 4 = 0 \text{ or } 1$$

Heuristic: Explain Discovery 4, i.e., why the remaining graphs on $n = 4$ are not self-complementary.
Heuristic: Consider the degrees of the vertices.
Discovery 10: A vertex of degree 0 in G becomes a vertex of degree n -1 in G^C. A vertex of degree n -1 in G becomes a vertex of degree 0 in G^C. In general, a vertex of degree k -1 in G becomes a vertex of degree n-k in G^C.
Fact: A graph with a vertex of degree n -1 is connected. A graph with a vertex of degree 0 is disconnected.
Fact: If G is self-complementary, then G and G^C are both connected or both disconnected.
Discovery 11: If G is self-complementary, then G has no vertex of degree 0 or degree n -1.

Figure 3: Episode 2 - Explaining Non-Examples (Part 1)

Heuristic: Generalize using explanation.
Conjecture: If G is self-complementary, then G is connected.
Proof: Let $G = \langle V,E \rangle$ be disconnected, and let x,y be in V. If x and y are in different connected components of G, x and y are connected in G^C by the path $\langle x,y \rangle$. If x and y are in the same connected component of G, for any vertex z in a different component of G, x and y are connected in G^C by the path $\langle x,z,y \rangle$. Thus if G is disconnected, then G^C is connected and G cannot be self-complementary. Therefore G is connected.

Figure 3: Episode 2 - Explaining Non-Examples (Part 2)

that set. During the first episode, the only ways the student could find self-complementary graphs was to construct and test all non-isomorphic graphs in order of increasing n. The research in the second episode repeatedly refines a new definition, a generator.

Episode 2 begins by trying to explain the "smallest" instance of failure, Discovery 2. Because the discovery is phrased in terms of n, an obvious guess is try to find an explanation in terms of n. Since the focus of attention (Figure 1) includes the formula for $max(|E|)$, the student computes it for $n = 2$ and $n = 3$, and notes that both values are odd. The computation of E^C in the definition of self-complementary suggests that Discovery 2 is an instance of a more general pattern, and generalizes it to Discovery 7. The generator for self-complementary graphs should thus avoid graphs for which $n(n-1)/2$ is odd, and skip all the graphs on 2 or 3 vertices entirely. Discovery 8 narrows this even further for graphs on n vertices; it considers only those with $m = n(n-1)/4$. Discovery 9 negates the characterization of non-self-complementary graphs to characterize partially the set of self-complementary graphs. A generator for such graphs would now only consider values for n in $\{1,4,5,8,9,12,13,\ldots\}$ with $m = n(n-1)/4$.

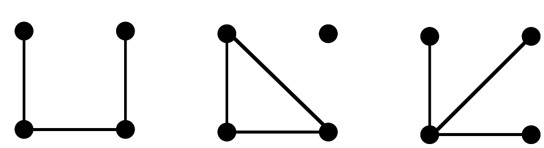


Figure 4 : Non-isomorphic Graphs on 4 Vertices with 3 Edges

The student now considers why, among the 11 non-isomorphic graphs on 4 vertices, only one is self-complementary. Discoveries 7 and 8 offer a partial explanation: for any graph on four vertices, there are at most 6 edges, and so only graphs with three edges should be tested. There are 3 non-isomorphic graphs with 4 vertices and 3 edges, shown in Figure 4. Since all these graphs have the same n and m values, the reason that they are self-complementary must involve some additional aspect of their structure. As a common graph descriptor, and one of the earliest definitions in Figure 1, the degree of the vertices is a reasonable candidate.

In Discovery 10, the definition of self-complementary (specifically, the reference to the concept of isomorphism) guides the search. The student considers the effect of isomorphism on the degree of a vertex, first for the extreme cases (0 and n -1), then in the general case. Since connectedness and disconnectedness are preserved under isomorphism, G and G^C are either both connected or both disconnected. The vertex degree observations in Discovery 10 are used to link connectedness with self-complementary graphs and produce Discovery 11. The generator for self-complementary graphs is restricted to graphs without vertices of degree 0 or n -1. (In a search for self-complementary graphs, this criterion alone eliminates 8 of the 11 graphs on four vertices and 21 of the 34 on five vertices.)

The link with connectedness now motivates a reexamination of the discovered examples. Thus far, all the discovered self-complementary graphs are connected, and every non-self-complementary graph on four vertices had at least one vertex of degree 0 or of degree n -1. The student generalizes these observations into a conjecture that all self-complementary graphs are connected graphs. The proof is actually of the contrapositive of the conjecture, which the student knows to be logically equivalent. A generator for self-complementary graphs should thus consider only connected graphs.

The actual discovery process is abbreviated here. One of the explored avenues went further into the degrees of the vertices. It discovered that for every self-complementary graph, the number of vertices of degree i +1 must be equal to the number of vertices of degree n-i. The stronger result, discovered and proved, was "Let the degree sequence of a graph be a vector d_i representing the number of vertices of degree i, for values of i ranging from 0 to n -1. This sequence must be a palindrome."

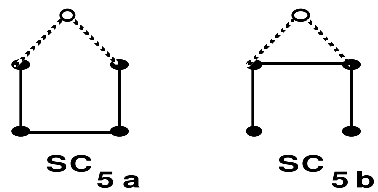
1.3 Episode 3: Explaining Examples

This section is to be read in conjunction with Figure 5. The second episode was able, while characterizing non-examples, to identify necessary, if not sufficient, properties of self-complementary graphs. The third episode attempts to detect and explain relations among examples, and to characterize them.

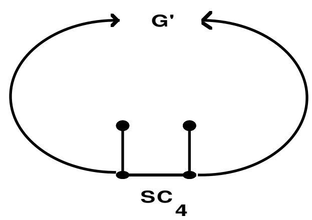
The directive to view self-complementary graphs as composed of other self-complementary graphs is directly attributable to a heuristic called constructional derivation, one which can be observed extensively in mathematics. Rather than a creative jump, Discovery 12 is simply a methodical application of this heuristic well-known to mathematicians. The observation is then generalized into a conjecture in a single hopeful leap.

To support this guess, its origin is exploited by mimicking its apparent construction: to construct a larger graph, one of the components is replaced with a larger self-complementary graph. Such a good, easily-found technique for constructing self-complementary graphs suggests that there are probably others, with similar proofs. A further, unfruitful search for construction techniques is

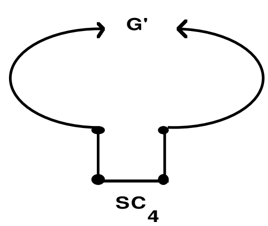
Heuristic: Find some similarity between examples SC_{5a} and SC_{5b}.
Heuristic: View the graphs as a composition of subgraphs. Both SC_{5a} and SC_{5b} can be viewed as $SC_1 + SC_4$:



Discovery 12: Each component of SC_{5a} and SC_{5b} in the decomposition is self-complementary.
Conjecture: Each self-complementary graph with $n > 4$ is composed of two self-complementary subgraphs.
Heuristic: Generalize the construction in the decomposition.
Construction: Replace the trivial self-complementary graph O_1 with any self-complementary graph G'. Combine G' with SC_4. Connect each vertex of degree 2 in SC_4 with every vertex of G', creating a new graph G.



Claim: G is self-complementary.
Proof: Consider a pair of vertices x,y with x in G' and y in SC_4. Let u' be an isomorphism map of $G' = <V',E'>$ to $G'^C = <V',E'^C>$. Extend u' to u with an isomorphism map of SC_4 to SC_4. If $(x,y) \in E$ then $(u(x),u(y)) \in E^C$ and vice versa. Thus u is an isomorphism from G to G^C, and G is self-complementary.
Construction: Replace the trivial self-complementary graph O_1 with any self-complementary graph G'. Combine G' with SC_4. Connect each vertex of degree 1 in SC_4 with every vertex of G', creating a new graph G.



Proof: Similar to the above.
Heuristic: Conjecture construction method is complete.
Conjecture: For any self-complementary graph G, there is a G' with 4 fewer vertices such that G' +SC_4 results in G.
End of Trace

Figure 5: Episode 3 - Explaining Examples

omitted here, but it results in the final conjecture, an assertion that the generation process delineated during these episodes is both correct and complete. In this third and final episode, the attempt to explain examples has resulted in a tentative characterization of all self-complementary graphs.

1.4 Analysis

The trace describes discovery in terms of fact-finding, observations, explanations, and generalizations based on them. Heuristics have been identified as probable catalysts for each step of the process. Some of the heuristics are the same as those of Lenat [14], Polya [18, 19], and Lakatos[13], and apply generally to mathematics, not only to graph theory. Although the trace is probably too "clean" (i.e., errors and forays along certain paths have been discarded), it offers some answers to the questions posed to the reader.

- What conditions might be necessary for an AI system to discover mathematical knowledge? Discovery is not merely the result of blind search. The minimal knowledge of Figure 1 (e.g., relevant concept definitions, and the formula for the computation of the maximal number of edges on n vertices) focused the student's attention and was a powerful guide to search. Neither is discovery conducted in a fixed space. New results, in the form of necessary conditions for the property of interest, were assimilated into the search procedures to restrict the search space.
- What role do examples play? Examples of graphs are referenced continually. Extremal examples are particularly helpful. Positive and negative examples are rich in information.
- What kind of tasks arise? General examples of graphs are constructed, in order of increasing "size," and such examples are tested for their ability to meet definitions. Discoveries arise as conjectures under inductive inference from these examples and as the results of tests upon them. The ability of two conditions (e.g., connectedness and self-complementarity) to coexist is a recurrent issue. There is a drive to seek similarities between examples of an idea, to seek alternative definitions (what mathematicians call *characterizations*) of a concept, to seek explanations for observations, and to couch them all in the descriptive language of the focus of attention. New concepts arise, e.g., SC_{5a} and numbers modulo 4, and are identified, described, and integrated into the knowledge base in progressively more sophisticated terms. (The concept "numbers modulo 4" is not novel but new in the sense that it arises in the course of discovery and is not known to be relevant a priori.) This gradual extension suggests the formation of a natural, partially-ordered hierarchy of descriptive languages that evolves during discovery. Finally, a very strong (probably trained) ability to do deduction and find proof is present, although such details are mostly suppressed in the trace.
- What heuristics appear to guide discovery? Some heuristics are clearly specialized for the domain of graph theory (e.g., "Consider the degrees of the vertices"), but others are not limited to graph theory (e.g., "One place to start is to try extreme cases") and not even to mathematics (e.g., "Generalize using explanation"). Even the singularly "creative" behavior in the trace (the perception of the graphs SC_{5a} and SC_{5b} as being similar according to a method of decomposition) is well accounted for.

The approach to mathematics described in the next section is designed to be responsive to these observations; it is not limited to graph theory.

2. A Framework for Looking at Mathematics

Although the thought experiment in the preceding section often uses a description (e.g., "connected" or "having a vertex of degree 0") rather than a name for a set, it repeatedly references sets of graphs and seeks relations among them. Because of their extensive appearance in the trace, this section formalizes sets of objects with relations among them and then uses that formalism to construct a context for mathematical research. The relation to feature abstraction, a traditional representational technique, is also shown.

Define a *universe* U to be any set of objects of interest, for example, functions or groups or graphs. A *property* P of the objects in U is simply a designated subset P of U; P contains exactly all the objects of U that "have" P. For example, if U is the set of all finite groups, P might be the set of all commutative groups, and be referred to as commutativity. By the usual extensional definition, properties P_1 and P_2 are equal if and only if $P_1 \subseteq P_2$ and $P_2 \subseteq P_1$. If U is finite, there are only finitely many properties of U; an infinite U is likely to be more interesting.

The discovery session is primarily about properties. Results appear as: "x is an example of an object with property P" or "x is an example of an object without property P." If mathematics is about properties, then conjectures, lemmas, and theorems in mathematics should regularly appear in forms such as the following:

- TYPE 1: If an object has property P, then it has property Q.
- TYPE 2: An object has property P if and only if it has property Q.
- TYPE 3: If an object has property P and property Q, then it has property R.
- TYPE 4: It is not possible for an object to have both property P and property Q.

It may be argued that a mathematician who reasons about P is mentally manipulating all of P. Thus *the mathematical study of U is an investigation of the properties of U and the relations between them.* A good representation for discovery in mathematics should, therefore, be a finite and flexible representation for P.

An aside about types of properties is appropriate here. As presented in this paper, the notion of property is boolean, i.e., an object in U either does (true) or does not (false) have P. Many mathematical properties, such as the number of terms in a polynomial or the number of vertices in a graph, are traditionally defined as numeric, i.e., mappings of U into some set of numbers. A boolean property partitions U into P and U - P. A numeric property partitions U when the property is a function defined on all of U. Rather than provide for such numeric properties, this paper restricts the definition to boolean properties that specify a number in their definition, for example, "graphs on five vertices."

A *characteristic* of an object is an ordered pair (P,b) where P is a property and b is either 0 or 1. If an object has property P, it is said to *satisfy* characteristic (P,1); otherwise it is said to satisfy characteristic (P,0). A *description* d is any set, finite or infinite, of such characteristics. An object *satisfies a description* if and only if it satisfies each characteristic in the description, and a description is *satisfiable* if and only if there exists some object that satisfies it. Thus the set of all objects satisfying a given description is itself a property.

Intuitively, the set of all properties in a description may be thought of as a mathematical context. If $d = \{(P_1,b_1),(P_2,b_2),\ldots,(P_k,b_k),\ldots\}$ is a (not necessarily finite) description, define $C = \{P_1,P_2,\ldots,P_k,\ldots\}$ to be its (not necessarily finite) *context.* In the context C, an object is described as having, or not having, some combination of those properties. The *C-characterization* of an object is the description with context C that is satisfied by the object. Such a characterization is always possible; for a specific object its C-characterization is the most detailed description composable within C. Although the properties in C need not be mutually exclusive, the set of all satisfiable C-characterizations partitions U into equivalence classes. Each such class is called a *C-class*. When C is fixed and finite, this technique is known as *feature abstraction* and $(b_1,b_2,\ldots,b_k)$ as a *feature vector.* The representational techniques described in the following sections actively support the construction of many different contexts, including infinite ones. The construction of such contexts, and the relations between them, is important.

3. Language and the Discovery Domain

The knowledge representation language for a discovery system both empowers and inhibits its performance. To the extent that the language is domain-specific, it can incorporate/support heuristics that speed search. Such specificity, however, may eventually backfire, and a system using a domain-specific language may be unable to follow a train of thought in an unanticipated direction.

On the other hand, general purpose representations and domain-independent heuristics have always proved weak; eventually, intelligent artifacts require domain-specific techniques to manage search. The history of AI is rife with periodic swings between domain-independent and domain-specific representation techniques. For a discovery program, the specificity requirement poses a particular difficulty: although the general domain of exploration is known in advance, the specific range of information, and the directions in which the system may drift during exploration, are, and probably should be, unknown to the system designer.

This inability to predict what needs to be represented directly affects knowledge representation for mathematical discovery; even the vocabulary in which information will be discovered is initially ill-specified, and yet a universal language will be powerless. One reasonable approach is to provide an *extensible* language, a limited domain-specific language that can be modified in a variety of ways to describe objects and relations as the need arises. The thought experiment confirms this need for language extensibility, and strongly suggests that an AI system for mathematical discovery cannot be limited to a fixed space guided by a fixed set of heuristics. The language must provide both notation for new concepts (e.g., SC_{5a}) and notation to control the complexity of knowledge already expressed (e.g., decomposition of self-complementary graphs).

4. The Construction of a Language for Graph Properties

This section applies both the lessons of Section 1 and the philosophy of Section 2 in the construction of a first context for graph theory, a language L_1. The reader interested primarily in the language supporting GT could proceed directly to Section 6. Those interested in applications of the thought experiment, formal grammars, and the evolution of a useful representation language are encouraged to read Sections 4 and 5 first.

GT's domain is graph theory. To delineate this domain we use three general texts: a classical development in elegant mathematical fashion [16], a broad overview of topics presented as definitions and theorems [11], and an algorithmic approach [2]. Together the content of these texts form an evaluation criterion for GT, a factual knowledge benchmark for discovery in graph theory.

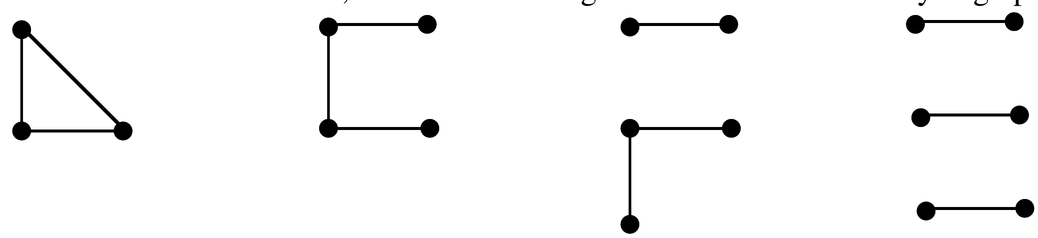


Figure 6: A Set of Graphs

A *graph property language* L is the set of all terminal strings generated by some formal grammar. Each terminal string in L is called an *L-expression*. (For example, "m = 3" might be a terminal string in language L.) The semantic interpretation of each L-expression is an *L-property* of a graph. (The semantic interpretation of "m = 3" is "all graphs with 3 edges.") Two L-expressions in a language L are *equivalent* if and only if their semantic interpretations are the same. (For example, in some language L, "m = 3" and the constructive description "add any number of isolated vertices to one of the graphs in Figure 6" might be shown to be equivalent.) Equivalence of semantic interpretation defines a set of equivalence classes on L-expressions. For example, in some language L, "m = 3" and the constructive definition above will both be in the same equivalence class of L-expressions. An equivalence class of L-expressions designates a subset of U, namely the set of all those graphs in U with the L-property interpreted semantically from the L-expression. (Among others, $G_1 = <\{1,2,3\},\{12,13,23\}>$ and $G_2 = <\{1,2,3,4,5\},\{12,34,25\}>$ satisfy "m = 3" and are in the subset of U designated by "m = 3".) The number of distinct equivalence classes of L-expressions is exactly the number of distinct L-properties.

As a first try at a context for mathematical discovery in graph theory, the language L_1 focuses on the similarities and differences among graphs due to their edge sets. L_1 is called an *edge-set language*, because each of L_1's four primitive symbols has a semantic interpretation as an edge set that appeared, in some fashion, in the thought experiment of Section 1: E represents the edge set itself, $I = V \times V$ is the set of all possible edges, $1 = \{(x,x) \mid x \in V\}$ is the set of all possible loops (a *loop* is a an edge from a vertex to itself), and $\varnothing$ is the empty set. There are two unary operators on any edge set S: reversal of the direction of all edges (denoted by S^{R}) and complementation with respect to I (denoted by S^{C}.) (Recall that complementation in the thought experiment was taken with respect to the non-loop edges, E_n ; here it is taken with respect to all edges, I.) Reversal is introduced in an attempt to distinguish directed from undirected graphs; complementation is chosen because of the significant role it played in the thought experiment. There are also two binary operations on edge sets, union and intersection. Finally, there is an equality relation (denoted as =) and an inequality relation (denoted as ≠) between any two edge sets constructed with the operators from the original four. These relations are chosen because of their repeated appearance in many fields of mathematics, and the prevalence of sets in the definitions of Figure 1. The formal grammar for L_1 on a graph $G = <V,E>$ is

symbol: E | I | 1 | ∅

term: *symbol* | $term^R$ | $term^C$ | *term* ∪ *term* | *term* ∩ *term*

expression: *term* = *term* | *term* ≠ *term*

Accept the convention of dropping parentheses whenever a construction would be unambiguous without them.

Some sample L_1 expressions are

$$E \cap 1^C \neq \emptyset$$

$$E \cup E^R = E$$

and

$$I = E \cup E^R \cup 1.$$

The interpretation of these expressions is in terms of the standard representation of the graph G by <V,E>. Thus the first expression is interpreted as "the set of graphs G such that the intersection of E and $\mathbf{1}^C$ is not the empty set" or "the set of graphs G such that the elements of E include some non-loops" or "the set of graphs G that include at least one non-loop edge." Similarly, the second is interpreted as "the set of graphs G such that, if an edge is in E, so is its reverse," and the third as " the set of graphs G such that every edge or its reverse is in E." Many L_1-expressions, however, are equivalent to others under this interpretation. For example, $E \cap 1^C = \emptyset$ is equivalent to $(E \cap 1^C)^R = \emptyset^R$ in its semantic interpretation.

If U is restricted to all undirected finite graphs, this edge-set language does not distinguish between E^R and E, or between E^{CR} and E^C. Thus there are fewer L-expressions for undirected graphs and therefore fewer L-properties.

Reversal and complementation can be restricted to symbols, rather than terms, without loss of expressive power [7]. That is, under this restriction, the same equivalence classes will be formed. Thus the following grammar will have the same L_1-properties, although its expressions are a subset of the first grammar's.

symbol: E | I | 1 | ∅ | E^R | E^C | E^{CR} | 1^C

term: *symbol* | *term* ∪ *term* | *term* ∩ *term*

expression: *term* = *term* | *term* ≠ *term*

This is the grammar used in [7] for L_1, where the following theorem is stated and proved:

> Although the language L_1 has infinitely many terminal expressions, there are only 12 distinct L_1-classes of undirected graphs and 24 of directed graphs. These classes may be defined using only four distinct properties for undirected graphs and five for directed graphs.

Unfortunately, the semantic interpretations of these L_1-properties are not particularly interesting to graph theorists; not one L_1-property is equivalent to any property definition cited in the index of any of the benchmark texts. Although the recursion of the grammar provides an infinity of L_1-expressions, their semantic interpretation results in an uninteresting, finite set of properties. The next step is to extend L_1 in an effort to meet one evaluation criterion for discovery, the target set of factual knowledge.

5. A Grammatical Hierarchy of Edge-Set Languages

Language L_1 of Section 4 was a formal language for graph properties, but an inadequate context for discovery. This section extends such formal languages into a grammatical hierarchy of progressively more expressive languages. Throughout this section, the operators and relations used in the extensions are chosen because graphs have a set-theoretic context and because mathematicians have found certain tools powerful, as exemplified by the thought experiment of Section 1. Because the semantic interpretation of L_1-expressions provided an inadequate vocabulary, sample expressions in each new language are used to monitor developing expressive power.

When the terminal strings of one formal grammar are a proper subset of the terminal strings of a second formal grammar, the language generated by the second is said to be *more expressive* than the language generated by the first. Any set of languages constructed in this fashion has a partial ordering based on the relation "is more expressive than." The languages so constructed and their partial ordering form a *grammatical hierarchy*.

Language L_2 is an extension of L_1 that permits the relations of cardinal equality (denoted as $\equiv$) and cardinal inequality (denoted as $<>$) between two edge sets. The new productions are

expression: *term* $\equiv$ *term* | *term* $<>$ *term*

The interpretation of an L_2-expression does not use integers in its property statements. L_2, like L_1, has only finitely many properties. Examples of properties that can be interpreted from L_2-expressions but for which no equivalent L_1-expressions exist include:

$$E^{CR} \equiv E$$

and

$$1 \cup E <> E^{C}.$$

Based again on the standard representation $G = <V,E>$, the first expression is interpreted as "the set of graphs G such that the reversal of the complement of the edge set has as many elements as the edge set does." The second is interpreted as "the set of graphs G such that there are not the same number of elements in the complement of the edge set as there are in the edge set and all the loops."

Language L_3 is an extension of L_2 that permits the relation of lesser cardinality (denoted as $<$) in addition to cardinal inequality ($<>$). The new production is *expression:* *term* < *term*

The interpretation of an L_3-expression still does not use integers specifically in its property statements, and there are still finitely many L_3-properties. Examples of properties that can be interpreted from L_3-expressions but for which no equivalent L_2-expressions exist include

$$E^{CR} < E$$

and

$$E < 1 \cup E^C.$$

Based again on the standard representation G = <V,E>, the first expression is interpreted as "the set of graphs G such that the reversal of the complement of the edge set has fewer elements than the edge set." The second is interpreted as "the set of graphs G such that there are fewer elements in the edge set than the number of vertices in G plus the number of non-loops in the complement of the edge set."

Recall that n denotes the number of vertices in the graph. Extend languages L_1, L_2, and L_3 to the languages L_{1n}, L_{2n}, and L_{3n}, respectively, by adding to each of them the productions:

integer: 0 | *integer* + 1

expression: n = *integer*

Each of the finitely many equivalence classes in L_1, L_2, and L_3 is split into finitely or infinitely many equivalence classes in L_{1n}, L_{2n}, and L_{3n}, respectively. Each of the languages L_{1n}, L_{2n}, and L_{3n} has an infinite number of properties and is therefore more expressive than its corresponding language L_1, L_2, or L_3.

Despite the addition of more properties from the thought experiment, none of the six edge-set languages discussed thus far is able to express more than a few of the properties in the benchmark texts. The construction of L_1^* tries a different approach and is more successful. Language L_1^* is an extension of L_1 that includes the symbol E*, the transitive closure of paths in E. E^* has a recursive definition:

$$xy \in E^* \text{ if } xy \in E \text{ or if } xp, py \in E^*$$

Thus *xy* is in E* for G = <V,E> if and only if there is a path $\langle x,y \rangle$ in G. Examples of properties that can be interpreted from L_1^*-expressions but for which no equivalent expressions in any of L_1, L_2, L_3, L_{1n}, L_{2n}, or L_{3n} exist include:

$$E^* \cap 1 = \emptyset$$

and

$$I = E \cup E^* \cup 1.$$

Using the standard representation G = <V,E>, the first expression is interpreted as "the set of graphs G such that no element of the transitive closure of the edge set is a loop" or "the acyclic graphs." The second is interpreted as "the set of graphs G such that every edge is in the edge set or represents a path in the edge set." L_1^* is more expressive than L_1 and has a finite basis.

L_1^* can express, for undirected graphs, a property such as "the set of graphs G such that every connected component is complete." Although such expressive power is an improvement on that of the preceding languages, semantic interpretations of L_1^*-expressions for undirected graphs are difficult to follow. For directed graphs, however, the properties (still finite in number) become a challenge to English grammar and, therefore, even less likely to be interesting to human mathematicians.

In summary, a grammatical hierarchy has been constructed; it appears in Figure 7, where an arrow from language L_a to language L_b means that L_b is more expressive than L_a. For finitely many properties, L-characterization (or feature abstraction) supports procedures to detect commonalities and differences in sets of objects, and to reason about classes of objects. The graph properties in L_1, L_2, L_3, and L_1^* are finite in number because set-theoretic rules (such as $I \cap S = S$ for any edge set S) simplify most of their terminal strings. Thus, although the expressions in languages L_1, L_2, L_3, L_{1n}, L_{2n}, and L_{3n} grow longer, after a certain point no new meanings are introduced. For discovery, the properties in those languages are uninteresting; even though they began with focus definitions from the thought experiment, the resultant relations and sets lack reference to the features, such as connectedness, that were so important in the thought experiment. L_1^* with its transformation E^*, comes closest to an interesting research context. Within the recursion of the grammar, E^* itself has a recursive definition. A more elaborate second application of recursion proves successful in the languages of the next section.

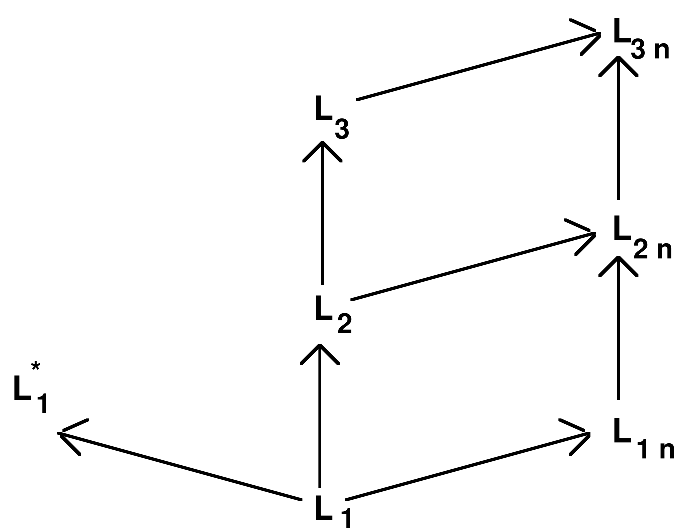


Figure 7: Orderings for Property Languages

6. The Recursive Languages

Although the edge-set languages were based upon the terminology of Figure 1, they did not meet the factual knowledge benchmark. This section builds upon the edge-set languages another hierarchy of representation languages, the R-languages. The R-languages satisfy many of the behavioral objectives and express much of the factual knowledge in the three texts specified as graph theory. A very simple R-language empowers GT's discovery in graph theory.

6.1 Origin and Rationale

The original motivation for the experiments described in this paper was a set of graphs collected by Charles Schmidt, a cognitive psychologist and computer scientist at Rutgers. Data in an experiment was described by graphs representing mental structures. All the graphs had the same number of vertices, but different edge sets. What, wondered Schmidt, might these graphs, i.e., these edge sets, have in common? The edge sets of the graphs distinguished them from each other, and yet he believed that some underlying principle created them. The question remains unanswered, but it was the catalyst for the work described here.

The representation languages for graph theory described in this section are based upon results from the preceding sections:

- In the thought experiment of Section 1, a conjecture (for example, that self-complementary graphs were connected) was initiated by the observation of examples. The proof, however, required the ability to reason abstractly about a class as a whole, not individual instances of it.
- Examples played a key role in the experiment. They familiarized the student with the property, initiated conjectures, and provided a testbed for them. Extreme examples were particularly useful.
- The constructional derivation in the experiment linked larger examples to smaller ones, organized the class, and suggested an alternative definition.
- Mathematical results are often expressible as relations between classes of objects, as in Section 2.
- L_1^*, the most appropriately expressive of the L-languages in Section 5, used recursion in its edge set description.
- Much of the thought experiment was devoted to constructing a generator definition for the class.

As a result, the languages developed in this section are designed to describe and reason about entire classes of graphs and the relations between them. Extreme examples, constructional derivation, and recursion appear in every property definition. Although reasoning about the relations between classes whose definitions are testers is reducible to reasoning about predicates (i.e., pieces of code), all but the simplest such programming tasks are beyond the current state of the art. In addition, the apparent drive toward a generator definition in the experiment suggests that perhaps testers are better reserved to examples, and that generator definitions support mathematical research. Thus the languages of this section are for generators of classes of graphs.

Recursion is used twice in the representation languages of this section: once in the grammar defining the language, and again in the interpretation of an expression. Mathematicians (see, e.g., [17], [20]) have long advocated recursion to capture the idea of an infinite set or process. GT appears to be the first program to employ the power of induction for class description. (See, however, its use in theorem proving [3] and, more recently, in shape description [15].)

Informally, an R-language expression for a property P is a concise, three-part string whose semantic interpretation is a specialized generator algorithm. That algorithm is describable as an input-free automaton (a *P-generator*) that, when started, generates a (possibly infinite) set P of graphs. As in Section 1, the P-generator so defined must be both correct and complete with respect to P.

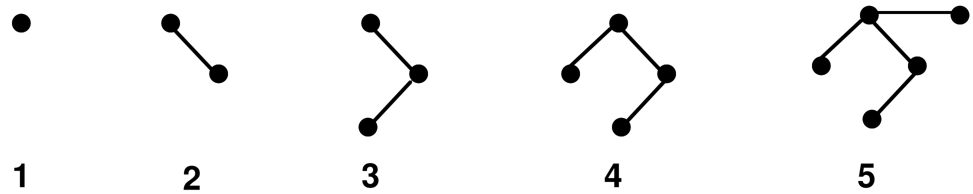


Figure 8: A Sample Run of TREE

The three key components of an R-language expression, < S, f, σ >, for a property P are

• S, the seeds, a set of start-up graphs

• f, the operation, a set of instructions to transform an input graph

• σ, the selector, a set of conditions governing the application of the operation

Informally, every seed is in P and every graph transformed by the algorithm produces another graph in P. The algorithm, if executed indefinitely and simultaneously on all its seeds, generates exactly P. As an example, consider the R-language expression for trees:

$$\text{TREE: } <\{K_1\}, A_{xy}A_y, [x \in V, y \notin V]>$$

The seed set S is $\{K_1\}$, the set whose only element is the complete graph on a single vertex. $\{K_1\}$ is, in number of vertices, a "minimal" tree. The operation f is $A_{xy}A_y$ ("branch from x to y"), meaning "add to the graph a vertex y and an edge from vertex x to y." The selector σ is "x ∈ V, y œ V," meaning "x is a vertex in the input graph and y is not." Figure 8 shows several iterations of TREE; each step is a branch that constructs a new graph. Each pictured graph is output by the algorithm and is a tree. The definition generates the infinite class of trees; it will never halt because bindings for the variables in s can be found on each iteration.

6.2 Definitions

This section provides the formal definitions for R-languages. Let L be a graph property language, F an operation language, and Σ a selector language. (Descriptions of these languages appear below.) Then <L,F,Σ> is an *R-language* whose terminal expressions (*R-expressions*) are triples of the form <S,f,σ>, where S is a terminal L-expression, f is a terminal F-expression, and σ is a terminal Σ-expression. The notation fσ is read "apply f subject to the constraints of σ." The recursive definition implicit in <S,f,σ> is

Every G ∈ S is in P.
If G ∈ P then fσ(G) is in P.
Nothing else is in P.

The R-language is declarative; the semantic interpretation is procedural. The semantic interpretation of <S,f,σ> is the following algorithm:

Accept G in S
Output G
Until σ fails **do**
$G \leftarrow f\sigma(G)$
Output G
Halt

The algorithm should be envisioned as running simultaneously on every graph in S. 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 *seed set language* L is itself a graph property language, typically an edge-set language from Section 4 or 5 of this paper. L must be able to describe the particular seed set S correctly and completely. In TREE, $\{K_1\}$ is the seed set. $\{K_1\}$ has the expression "$E^C \cap 1^C = \emptyset$ and $E = \emptyset$" in L_1. ($E^C \cap 1^C = \emptyset$ stipulates that no non-loop edges are missing from the graph, and $E = \emptyset$ that there are no edges, loop or non-loop, in the graph. Together they imply that the only possible edge in the graph is a loop, i.e., the graph has only one vertex.) Thus $\{K_1\}$ has a terminal expression in L_1.

The grammar for an *operation language* F has terminal symbols {f,g,h,…}, each representing some primitive transformation operator. The only non-terminal symbol is α. The productions are

$\alpha \rightarrow \alpha\alpha \mid \alpha + \alpha$
$\alpha \rightarrow f \mid g \mid h \mid \ldots$

The semantic interpretation of *fg* is "apply *g* followed by *f*." The semantic interpretation of $f + g$ is "apply exactly one of *f* and *g*." It is sufficient to define an operation language F by its terminal symbols. The operation language F_1 is defined by the terminal symbols $\{A_{xy}, A_y\}$, where A_y means "add vertex x to the graph" and A_{xy} means "add edge xy to the graph." $A_{xy}A_y$ is a terminal expression in F_1.

The *selector language* Σ is generated by a formal grammar too. Σ_1 will serve as an example:

$\alpha \rightarrow \alpha,\alpha \mid$ *vertex sign* V $\mid$ *edge sign* E
vertex $\rightarrow$ x $\mid$ y $\mid$ z $\mid v_1 \mid v_2 \mid v_3 \mid \ldots$
edge $\rightarrow$ (*vertex*,*vertex*)
sign $\rightarrow \in \mid \notin$

An example of a terminal Σ_1-expression is " $x \in V, y \notin V$," the selector for TREE. More elaborate Σ-languages describe such restrictions as relations among vertices and edges, the degree of a vertex, and the maximum degree of any vertex in the graph.

From the preceding discussion, it is clear that the expression

TREE: $<\{K_1\}, A_{xy}A_y, [x \in V, y \notin V]>$

lies in the R-language $<L_1,F_1,\Sigma_1>$, since $\{K_1\} \in L_1$, $A_{xy}A_y \in F_1$, and $[x \in V, y \in V] \in \Sigma_1$.

The grammars facilitate the definition of hierarchies of operation languages, seed set languages, and selector languages. The partial ordering of those hierarchies defines a partial ordering on the language triples. Thus the R-languages, like the languages of Sections 4 and 5, form a grammatical hierarchy.

6.3 Relevant Issues

The reader is likely to have many questions about the R-language representation by now. This segment of the paper attempts to anticipate and respond to them.

Isn't the semantic interpretation algorithm for generating examples from $\langle S,f,s \rangle$ non-deterministic?

Yes. In any R-language expression $\langle S,f,\sigma \rangle$ for property P, although all of S must be used to generate all of P, any element of S may be used to start the P-generator. Also, on each iteration there may be many bindings valid under σ; any one may be used.

Isn't the semantic interpretation algorithm for generating examples from $\langle S,f,\sigma \rangle$ ambiguous?

Yes again. For any G in P there are probably many different iteration sequences (possibly beginning with different seeds) that generate G. These ambiguities are quite deliberate. In addition, it is possible that more than one R-language expression generates the same P. These different expressions reflect the ability of the language to provide alternative characterizations of a given property.

Doesn't all that uncertainty and redundancy affect the algorithm's performance?

It does, but that is insignificant because a P-generator is rarely executed. It is a concise expression for a set of graphs, an expression amenable to the construction, modification, and possibly the verification of members of the set. For application, the only performance criteria are correctness and completeness.

Is an R-language property always satisfiable?

If the seed set S is non-empty, an R-language property is always satisfiable, because $S \subseteq P$. If S is empty, the property is never satisfiable, because the algorithm cannot execute.

Is every R-language expression a valid graph property?

Subject to certain restraints, an R-language expression is a valid graph property. If $\langle S,f,\sigma \rangle$ is the expression for P, it must begin with, and produce, only graphs. Therefore, S must be a set of graphs, and σ must make certain that f alters V or E only in ways that preserve $\langle V,E \rangle$ as a graph. Specifically, f may only delete vertices of degree zero, and f may only add edges between vertices already present in the graph. These restrictions prevent a graph from degenerating into just any pair of ordered sets. Obviously, f is never permitted to change V or E into anything other than a set.

How does one prove correctness and completeness for an R-language expression?

In most cases, proofs will require facts from graph theory. To demonstrate that all the graphs generated by the algorithm interpreted from $\langle S,f,\sigma \rangle$ are in P, one must show that $S \subseteq P$ and that transformation under $f\sigma$ preserves membership in P. Under these circumstances, $\langle S,f,\sigma \rangle$ will be correct. Except for intuitively simple properties, completeness is generally more difficult to show. One successful technique is to show that there is an *inverse*, denoted $\langle S,f^{-1},\sigma^{-1} \rangle$, that is correct.

The inverse is an algorithm that reverses the construction process. The situation is pictured in

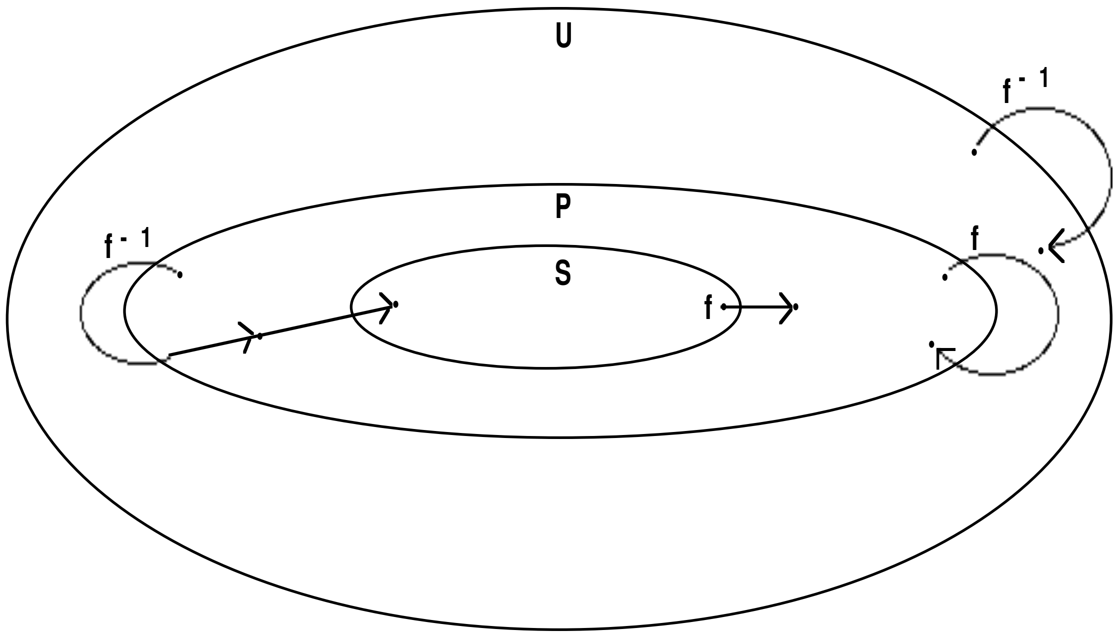


Figure 9: The Behavior of f and f^{-1} within the Set U of All Graphs

Figure 9. For P interpreted from $\langle S,f,\sigma \rangle$, f maps $G \in S$ into P, and f maps P into P. For the inverse $\langle S,f^{-1},\sigma^{-1}\rangle$, f^{-1} maps $G \in P$ into P and eventually back into S. Assume that f^{-1} is correct, i.e., f^{-1} only maps $G \in P$ into $f^{-1}(G) \in P$. Let $G \in P$. Since f^{-1} is defined on G and f^{-1} is correct, there exists some sequence of applications $f^{-1*}(G)$ that is a "trail" back to some seed graph $H \in S$. Inversion of these applications into $f^{*}(H)$ creates a trail from H to G. Thus G is reachable via f, and $\langle S,f,\sigma \rangle$ is complete. As an example, the inverse for

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

is

$$\text{TREE}^{-1}\text{: } \langle \{K_1\}, D_yD_{xy}, [x,y \in V, xy \in E, x \neq y, d(y) = 1]\rangle$$

(Observe that the inverse TREE^{-1} requires a more expressive Σ-language than TREE does, one which is able to specify the degree of a vertex.) TREE^{-1} gradually dismembers any tree. TREE^{-1} finds, on each iteration, distinct, adjacent vertices x and y in the graph, such that y is of degree one, and then deletes the edge xy and the vertex y. TREE^{-1} strips one branch at a time from any tree, until the original graph is reduced to K_1. (Read Figure 8 from right to left for an example of this destructive process.) If, however, the input graph to TREE^{-1} is not a tree, TREE^{-1} will reduce that graph's acyclic connected components to isolated vertices, and the remainder of its components to biconnected ones; the non-tree graph will never be transformed into K_1. This technique for proving completeness will work, of course, only for graph properties that have inverses. For those properties that do not (for example, one defined by a recursively enumerable, non-recursive set), completeness is generally proved by a demonstration that any graph with P can be constructed under a finite number of iterations of $<S,f,\sigma>$.

Given the expression $<S,f,\sigma>$ for P, under certain conditions it is possible with heuristics to construct automatically the inverse expression $<S,f^{-1},\sigma^{-1}>$ for P. Those conditions are discussed extensively in [7]. The algorithm associated with $<S,f^{-1},\sigma^{-1}>$ is a P-tester, and the automated construction of a P-tester from a P-generator is called *inversion.* The inversion from TREE to $TREE^{-1}$ is detailed in [7].

What data structure is used for graphs?

R-language expressions are theoretical descriptions designed to be independent of any data structure for graphs. In an implementation that generates examples, however, the variables in the R-language expressions must be bound to actual vertices and edges according to σ. The Σ-language may thus suggest one or more economical, and possibly unorthodox, structures. (For example, vertices and edges may be arranged in lists of biconnected components, or even as sets of cycles.) In the implementation, GT uses adjacency lists, not an edge-set language, to describe seeds, because the adjacency lists facilitate matching on binding criteria such as $x \in V$ and $d(x) = 1$.

What sort of automaton is a P-generator?

The P-generator for $<S,f,\sigma>$ has an obvious three-tape Turing machine counterpart. The first tape contains the program to modify the input graph (the information contained in f and σ) and space for computations. As f and σ are defined, they are certainly finite and encodable for a Turing machine. The second tape is for S. As long as the set S is finite or recursively enumerable, the second tape either lists S or encodes the Turing machine that generates S. If S is not finite or recursively enumerable, the R-language expression is implemented by a Turing machine with an oracle that decrees whether or not a given graph is in S. The third tape is for the output, that is, the set P. The Turing machine's program instructs it to select any graph G_1 from S, modify it into some graph G_2, output G_2, and then recurse, to modify G_2 on the next iteration. (It may be helpful to imagine a set of machines, at this point, one for each element in S, all operating in parallel. S is recursively enumerable and the non-determinism of the algorithm is not an issue for a Turing machine [12].) Thus, if there exists an R-language expression for graph property P with some recursively enumerable S, P is recursively enumerable. If there exists an R-language expression for P and an oracle for S, P is recursively enumerable relative to S.

What are reasonable complexity measures for P-generators?

Theoretically, one could measure the complexity of a P-generator by the (time and space) resources required to construct a finite P, or to construct some preselected set of examples X from P, or even to perform a single iteration other than the first. In all of these tasks, the selection of a seed can be done in constant time; it is the matching dictated by the selector that should be the key factor. The time required to find bindings for the operator variables meeting the selector descriptions depends upon the way, or ways, graphs are stored. Thanks to the adjacency list representation for specific graphs, GT's uniform procedure for actual example generation does not degrade performance, except for isomorphism testing. On the relatively rare occasions when a distinct set of examples of P is required, the generator's redundancy demands that the examples be stored and compared for distinctness. This isomorphism testing is currently quite primitive and slow; it is under revision. At no other time is the complexity of the P-generator ever an issue.

Is every graph property of interest to mathematicians recursively enumerable?

A graph property defined on graphs of no more than n vertices will be a finite set and, therefore, recursive. For graphs of unbounded size, however, it is possible to construct a non-recursively enumerable set of graphs. For example, let Q be a non-recursively enumerable set of integers. Then

$$Q^* = \{G = \langle V,E\rangle \cdot G \text{ is a graph, } |V| \in Q\}$$

is a non-recursively-enumerable graph property. Such properties, although they exist, are in some sense pathological. A graph theorist defines a property and then studies it. Study seems to require the ability to decide (determine the presence or absence of) the property by some computation, typically a tester. Most graph properties of intrinsic interest to graph theorists are probably not only recursively enumerable, but also recursive. The criteria for discovery established earlier in this paper focus on properties that people actually study in the course of research in graph theory (what [5] calls "natural classes"), not properties that they could study if they were so inclined.

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

Table 1: Graph Properties Studied under Recursive Generation

Is every graph property of interest to mathematicians expressible in some R-language?
Table 1 lists 42 graph properties believed to be formulated in R-languages in [7] correctly and completely. These properties were a cross-section selected from the indices of the benchmark texts. The properties vary both in their conceptual difficulty and in the length of their R-language expression. Those in italics have been encoded for or discovered by GT during actual runs. Many other properties with well-known equivalent definitions (such as "line graph") are also clearly expressible in an R-language. All these properties are empirical evidence that R-languages should be adequate. There is no formal proof.
Is the R-language representation applicable to labeled graphs and networks?
Extensions to R-languages [7] have included a register used as a counter, and labels on both vertices and edges. Some of the properties of Table 1 are written in those extended languages. Thus R-languages appear applicable to labeled graphs and networks. In addition there are R-languages that require no matching at all, i.e., for which Σ is empty.
Why focus on graph property languages to the exclusion of, say, graph relation languages?
The focus here has been on graph property languages because the thought experiment demonstrates an underlying preoccupation in mathematics with sets of objects. There have been, of course, alternative knowledge representations postulated for graph theory (see Section 7), but no published reports of their implementation. There is no claim that alternative representations are less valid, only that none have yet displayed GT's implemented success.
Where are the logical connectives and quantifiers that provide a language with power?
As discussed in Section 6.4 below, R-languages have immediate access to procedural equivalents for connectives and quantifiers. The boolean "and," for example, is available via merger. Quantified descriptions such as "having at least one spanning forest" are written constructively, by generating an expression that produces such graphs.

6.4 Applications for the R-language Representation

This section is an overview of the implementation of one fairly simple R-language in a program named GT. The description appears here only to support the claim that the R-languages are responsive to both kinds of evaluation criteria, behavioral objectives and factual knowledge. The reader seeking implementation details is referred to [8].

GT manipulates graph theory properties via their $\langle S,f,\sigma\rangle$ representations. Given an R-language expression $\langle S,f,\sigma\rangle$, GT can generate any number of arbitrary graphs with that property. There is a single, uniform procedure that accepts *any* $\langle S,f,\sigma\rangle$ definition and produces these examples. Similarly, there are uniform procedures, notably subsumption and merger, which support the proof of the four theorem types identified in Section 2.

Given expressions $\langle S_1,f_1,\sigma_1\rangle$ for P_1 and $\langle S_2,f_2,\sigma_2\rangle$ for P_2, it is possible with heuristics to detect that P_1 is a special case of P_2, i.e., that P_1 is a subset of P_2. This is called testing for *subsumption*, and P_1 is said to *subsume* P_2. GT has a uniform subsumption testing procedure for any pair of <S,f,σ> definitions. Subsumption testing shows, for example, that ACYCLIC subsumes TREE. Repeated applications of this procedure facilitate GT's construction of a relational hierarchy of graph theory knowledge. It is also possible to detect the equivalence of two properties: if P_1 subsumes P_2 and P_2 subsumes P_1, then P_1 and P_2 are equivalent. GT uses this method to detect equivalent property definitions.

Given expressions $<S_1,f_1,\sigma_1>$ for P_1 and $<S_2,f_2,\sigma_2>$ for P_2, under certain conditions it is possible with heuristics to *merge* the expressions into a single new R-language expression for $P_1 \cap P_2$. GT's merger procedure does precisely this. The resultant algorithm is a correct and complete generator for $P_1 \cap P_2$. GT's merger procedure has, for example, enabled it to construct expressions for "trees-with-an-odd-number-of-vertices" and "acyclic-and-connected-graphs." It is also possible, under certain circumstances, to demonstrate within the merger procedure a fundamental incompatibility between two properties, i.e., $P_1 \cap P_2 = \emptyset$. During its attempt to construct an expression for "odd-regular-graphs-on-an-odd-number-of-vertices," GT observes that an integer cannot be both odd and even, and declares, correctly, that there cannot be any such graphs.

Given a set of R-language expressions for graph properties, it is possible to conjecture relations among the properties based on similarities or differences in the expressions themselves. GT bases its conjectures on such details as degree of similarity between two seed sets, seeds of one property that are known to have the other property, and degree of similarity between two operators.

Given an R-language expression for a graph property, it is possible to modify that expression to produce new properties from the known one. GT currently has three uniform procedures for this: specialization, generalization, and merger. GT uses specialization to discover (construct) definitions for, among others, "connected" and "acyclic" from the definition for graph. GT models the appropriate insertion of new information into a preexisting knowledge base by generalizing that information until it can be linked into GT's relational hierarchy of graph properties. For example, the definition for stars is generalized until it is eventually recognized as a special case of connected graphs.

If the heuristic procedures alluded to above are themselves demonstrated correct, then it is possible to prove theorems on R-language expressions using those procedures. The formulations from Section 2 and their GT procedural equivalents appear in Table 2.

To prove	**Show that**
If a graph has property P, then it has property Q.	Q subsumes P.
A graph has property P if and only if it has property Q.	Q subsumes P and P subsumes Q.
If a graph has property P and property Q, then it has property R.	R subsumes the merger of P and Q.
No graph can have both property P and property Q.	The merger of P and Q is empty.

Table 2: GT's Procedural Equivalents for Theorem Proofs

GT uses R-language expressions and its subsumption and merger procedures to successfully conjecture and prove, among other theorems, the following:

• Every tree is 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.

The descriptive ability of an R-language lies in the number and nature of the primitive operators permitted in f and of the selector descriptions permitted in s. As the set of such operators and descriptions is extended, a lattice of descriptive languages is constructed. As Section 5 demonstrates, certain representation languages separate the universe U of all finite graphs into more classes than others. (For example, L_1 distinguishes 12 classes of undirected graphs, L_2 106.) Thus, certain graph properties are simply not expressible at certain levels of a grammatical hierarchy. It is interesting to identify the *floor* for a graph property P, informally the weakest possible language in a grammatical hierarchy that can represent P. Because the hierarchy is partially ordered, a property can have more than one floor.

The floor of a graph property has two interesting applications, a natural ordering and a resource metric. First, an extended R-language offers additional alternatives, and should have greater expressive power. A "more complex" language L_c distinguishes more classes of graphs than a "simpler" language L_s. (In the terminology of Section 4, there exist L_c-characterizations that are not representable in L_s but the union of whose equivalence classes is representable in L_s.) Tabulation of the new graph properties expressible/discoverable within each extension could provide a natural pedagogical order for graph properties. Second, operations within an extended R-language are likely to require more computer resources; the floor of a property should be useful in estimating the resources required for computation with that property. Within the discovery framework described here, plans exist to extend GT's R-language for the representation of directed graphs and, eventually, for labeled graphs. These extensions will provide a testbed for the study of performance under representational shifts, assuming that a language does not have access to more efficient data structures or procedures simply by virtue of its extended expressive power.

The key to GT's most interesting specializations (discovered definitions), those involving additional descriptions in σ, is the language in which those descriptions may be written. Banerji [1] writes eloquently about learning in a growing language. Utgoff's work with LEX [26], a machine learning system to discover heuristics for the application of symbolic integration rules, is a case in point. LEX was unable to formulate a pattern matching description for a rule it discovered because its knowledge representation language for integrands had no expressions for "odd" and "even." Unless the σ-language is extensible, its associated R-languages may not be able to access many interesting ideas. Ways to automate the extension of the σ–language for GT are under investigation.

GT's success validates the use of R-language expressions as definitions. These generator definitions prove to be more transparent and flexible than the more traditional feature vectors. Like

META-DENDRAL [4], GT can afford exhaustive search because its representation is highly controlled. Its property definitions support most behavioral objectives without recourse to empirical data.

6.5 A Mechanized Thought Experiment

GT is implemented on a Symbolics 3675 in Symbolics Common Lisp. The screen is divided into a display area and a smaller sketch pad where GT draws example graphs one at a time as it generates them. The sketch pad pictures go by quickly, but often convey what is, anthropomorphically, a "train of thought." When the program executes, the user is offered several options. We describe here an example of a mechanized thought experiment, a GT run. The user starts GT up with an input knowledge base. GT runs and discovers theorems.

The knowledge base of seven graph properties initially consisted only of property names and expressions its R-language: connected, acyclic, tree, chain, star, odd-number-of-vertices, and "equiv-connected," an equivalent definition of connected with a different R-expression. GT displays those definitions and then begins to conjecture about possible subsumption relations. All of these conjectures appear on the screen , and are then examined and reported upon one by one. Actual output at this point includes:

> I just tried to prove that the property P[TREE]
> subsumes the property P[CONNECTED].
> The result was NIL.

and

> I just tried to prove that the property P[ACYCLIC]
> subsumes the property P[TREE].
> The result was T.

From the positive subsumption results, the program makes conjectures about equivalence relations, and goes on to explore them. For example, two equivalent input definitions are discovered now:

> I just tried to prove that the property P[EQUIV-CONNECTED]
> is equivalent to the property P[CONNECTED].
> The result was T.

and some reasonable guesses are rejected:

> I just tried to prove that the property P[ACYCLIC]
> is equivalent to the property P[TREE].
> The result was NIL.

At this point GT warns us that equivalent property definitions will now be ignored, i.e., EQUIV-CONNECTED will no longer figure in its investigation. Next GT formulates and investigates some property mergers. The pictures on the sketchpad become particularly interesting at this point, as examples of the merger concepts appear. For example:

> I am going to call the new property, created by the merger of
> P[ODD-NUMBER-OF-VERTICES] and P[TREE],

ODD-NUMBER-OF-VERTICES-MERGED-WITH-TREE.

The example on the sketchpad at this moment is shown in Figure 10.

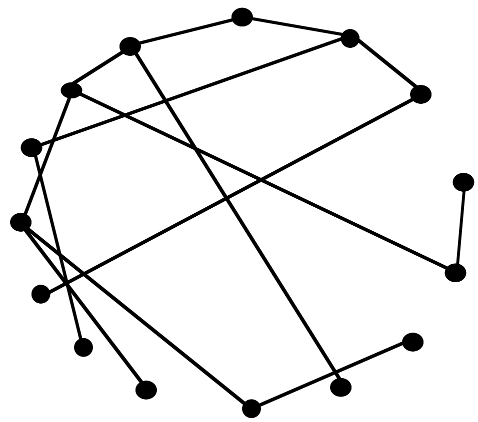


Figure 10: A tree with an odd number of vertices, generated by GT.

Some messages reference previous results during this stage, such as:

The property P[ACYCLIC] subsumes the property P[TREE]
and therefore the merger is simply P[TREE].
No new property was discovered.

Sometimes GT is uncertain of the new seed for a merger. Then it sends a cautionary message describing its choice, for example:

I am taking the smallest common graph
(G[ODD-NUMBER-OF-VERTICES-10]) as
a presumed seed for P[ODD-NUMBER-OF-VERTICES] and P[STAR].

In this case G[ODD-NUMBER-OF-VERTICES-10] is the tenth generated example of the property ODD-NUMBER-OF-VERTICES, and the smallest example of that property that was also found to be a star. (A *star* is a connected graph in which one vertex is adjacent to all the others, and every other vertex is of degree 1.) The graph, sketched on the pad, was indeed the correct seed; it appears in Figure 11.

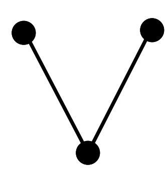


Figure 11: The smallest star on an odd number of vertices.

Failures have an interesting rationale at this point. For example:

This conjectured property had very few (1) examples.
It was rejected as trivial.

I just tried to merge the properties P[CHAIN] and P[STAR].

Now GT looks for additional subsumption relations using the new concepts formed by merger, and then uses those to conjecture and prove new equivalence relations. For example,

I just tried to prove that the property P[TREE] is equivalent to
the property P[CONNECTED-MERGED-WITH-ACYCLIC].
The result was T.

Finally GT displays its original and discovered knowledge about the seven original properties in its knowledge base. What, at the beginning of the run, took only a few lines, now requires seven screenfuls, simply listing the subsumption and equivalence investigations and their outcomes, omitting proofs and example graphs.

7. Discussion of Results

This paper recounts three experiments addressing issues in knowledge representation for mathematical discovery: the thought experiment, the edge-set languages, and the R-languages. There are lessons to be learned for mathematical discovery from each of them.

The thought experiment is a paradigm for mathematical discovery. It is an original and illuminating example of the discovery process, one which contains extensive information on behavioral objectives and ways to achieve them. (It also offers insights into the design of the memory system and the processing strategy [25].) In particular, the thought experiment highlights the significance of examples, the use of both inductive and deductive methods, and the role of heuristics.

The edge-set languages are reasonable attempts at capturing the language of the thought experiment; they focus on the edges, the smallest unit of information in the trace that describes a relation within graphs. The resultant class descriptions, although they echo the "basis language" of the experimenter, ultimately fail to capture the graph properties of interest to mathematicians. Thus it is not enough, in this case, to incorporate the superficial description, i.e., language, of a discovery protocol; something more is required.

The R-languages base their class descriptions on extreme examples, recursion, and constructional derivation. An expression in any one of these languages contains all these elements and is interpretable as a generator definition for a class of graphs. Although the existence of a definition of the form $\langle S,f,\sigma \rangle$ for every property P remains an open question, the expressive power of the R-languages seems adequate for the factual knowledge in the benchmark texts. GT, an implemented version of one simple R-language, demonstrates that the generator definitions are highly flexible and subject to heuristics that effectively model many of the behavioral objectives of the thought experiment.

There has been, of course, other research on representation for graphs and attempts to study graph theory with the assistance of a computer. Galperin [9] represented graphs by "cloning and gluing". The resultant algorithms generated classes of graphs efficiently, but they lacked the R-languages' uniformity of algorithmic representation and were not implemented. Other relevant, unimplemented efforts include Rosen's grammar-like systems on directed graphs [23], Wegman's path language for directed graphs [27], Proskurowski's notationless description of recursive labelings to find shortest paths in undirected graphs [22], and Yannakakis' transformations on an arbitrary graph to attain a given property [28]. Pratt and Friedman implemented a transformational language for processing directed graphs [21], and Dorin implemented a Graph Programming Language machine to produce sequences of directed graphs based on the algebraic theory of graph grammars [6]. The latter manipulated individual graphs within classes, but did not perform research in graph theory. The only behavioral objective visible in any of this related work is the efficient generation of examples; the expressive power of the representations varies broadly. There are currently only two other programs that attempt mathematical research, as defined here, in any subdomain: AM in set theory and number theory [14] and IL in Conway numbers [24].

The remainder of this paper addresses the relations between GT and the thought experiment, and evaluates the significance of the subdomain results for mathematical discovery.

7.1 GT and the Thought Experiment

GT demonstrates the power of one fairly simple R-language. All GT's properties are defined in the same representational language, and all its procedures are uniform, i.e., they are property-independent. Particularly in the mode where it begins only with the definition of graph, GT simulates mathematical discovery. How does GT do on the behavioral objectives of the thought experiment, however, and how would it do on the specific thought experiment task?

GT simulates the overt behaviors of the thought experiment, but its underlying procedures do not necessarily match the ones a reader of the trace might attribute to its student. In the course of execution, GT regularly generates examples of all of its definitions. This generation is, like the student's, in order of increasing "size." Given appropriate tester definitions (LISP predicates), GT can test whether or not examples meet them, but GT usually finds testing unnecessary. GT's discoveries arise as conjectures under inductive inference, but the conjectures do not rely solely on examples (as AM's did); they are also based on definitional form. GT simulates interest in the ability of two conditions to coexist by planning and executing mergers. Using subsumption and merger, GT is able to discover relations among sets of graphs; within the perspective of Section 2, this simulates the search for relations among examples. GT models the search for characterizations by equivalence testing of definitions. GT, admittedly, does not present its work as a search for explanation; its present control structure might be paraphrased as "learn everything." GT really does work in the descriptive language of the focus of attention, but that language is hidden within the component languages for f, for S, and for σ. GT is able to construct proofs, but all of them follow one of a small set of patterns. Remarkably, that small set is quite robust.

GT has not run the thought experiment itself, because the current expression for self-complementary is constructed in a more complex R-language than the implemented one. (In addition, the constructive definition for self-complementary that appears as the final conjecture in Figure 5 has not yet been shown complete, although there is no known counterexample.) Thus the following discussion is based only on a working knowledge of the system, the relevant R-language, and the proposed definition. Episode 1 of the thought experiment generated examples of a property; GT does that expertly, without ever drawing a non-example. GT keeps a list of examples for each property and attempts to minimize redundancy with a primitive isomorphism tester (currently under refinement). Episode 2 explained why certain graphs were not examples of a property. When GT encounters a new concept, the system immediately attempts to integrate the concept into its hierarchy of known concepts. One way GT does that is to modify the new concept definition and look for relations among the modified definitions and the known ones. If GT were given or discovered the properties "having 2 vertices," "being disconnected," and so on, its relentless attempts to incorporate the new concept should eventually produce the results in Episode 2. GT cannot, however, produce a definition for the negation of a concept from the concept itself. Instead, GT either reports that its (very fast, clever, and powerful) attempts to show that a graph is generated by a definition have failed (which is very good evidence but not certainty), or it proves that two classes of graphs have no intersection, which shows that the graphs with one property do not have the other property. The precision with which the student in the trace hones in on the relation to connectedness may be exaggerated by the artificiality of the protocol, and certainly would not be within GT's current skills. Finally, Episode 3 seeks a constructional derivation definition, a generator instead of a tester. GT can use LISP predicate testers, but its orientation is certainly toward generators. Although some results in [7] indicate that inversion, to transform generators into testers and vice versa, is possible under certain circumstances, automated inversion would require the implementation of a hierarchy of languages and is beyond GT's current scope.

7.2 Lessons for Mathematical Discovery

This paper is directly relevant to research in mathematical discovery, and is not limited to graph theory. The specific principles follow.

- A mathematical discovery system should be held to two kinds of criteria: a set of behavioral objectives, which are independent of any mathematical subdomain, and a target set of factual knowledge.
- A thought experiment is a protocol paradigm for the empirical study of mathematical discovery. It is a rich source of mathematical research behaviors for simulation and of subdomain-specific terminology and methods.
- The view of mathematics as properties (classes of objects) and relations among those properties is consistent with both sets of criteria and supports automated mathematical discovery.
- The language in which discovery is described is not completely reflective of the processes that underlie it.
- The efficient, uniform, and flexible representation of large homogeneous classes of objects is necessary to any mathematical discovery system. Recursive generator definitions, as representations for object classes, can be manipulated to manifest such qualities more successfully than testers.

- Examples play a key role in the discovery process: they form the basis for inductive inference, support or discount conjectures, and offer bounds for object classes.
- Constructional derivation from simple examples combines the power of recursion with the specific knowledge to be gleaned from representatives of an object set.
- The balance between general and domain-specific techniques is particularly crucial in the design of languages for discovery systems.
- Although discovery may begin with a language of limited expressive power, that language must be extensible. Effective search during discovery demands a hierarchy of increasingly expressive languages.

GT is a programmed demonstration that these principles are effective. Its knowledge representation is supported by the thought experiment and validated by its performance. Even its simple R-language supports multiple mathematical discovery processes: correct example generation, the construction of new properties, and the conjecture and proof of mathematical theorems. GT's most significant omission is counterexamples (see, e.g., [13]); they are the primary target in GT's current development.

Acknowledgements

The authors thank Saul Amarel, Tom Mitchell, Marvin Paull, Rick Statman, Virginia Teller, and Ann Yasuhara for their insightful questions and discussions.

Bibliography

[1] R. Banerji, *Theory of Problem Solving: An Approach to Artificial Intelligence*, American Elsevier Publishing: New York, 1969.

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

[3] R.S. Boyer and J.S. Moore, *A Computational Logic,* Academic Press: New York, 1979.

[4] B.G. Buchanan and T.M. Mitchell, "Model-Directed Learning of Production Rules," in *Pattern-Directed Inference Systems*, edited by D.A. Waterman and F. Hayes-Roth, Academic Press: New York, pp. 297-312, 1978.

[5] J.G. Carbonell, R.S. Michalski, and T.M. Mitchell, "An Overview of Machine Learning," in *Machine Learning: An Artificial Intelligence Approach*, edited by R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, Tioga Publishing: Palo Alto, pp. 3-23, 1983.

[6] P.M. Dorin, "Aspects of the Implementation of Sequential Graph Rewriting Systems," Ph.D. dissertation, Department of Computer Science, University of California, Los Angeles, 1982.

[7] S.L. Epstein, "Knowledge Representation in Mathematics: A Case Study in Graph Theory," Ph.D. dissertation, Department of Computer Science, Rutgers University, 1983.

[8] S.L. Epstein, "Learning and Discovery: One System's Search for Mathematical Knowledge," in *Computational Intelligence* vol. 4-1, pp. 42-53, 1988.

[9] H. Galperin, "Succinct Representation of Graphs," Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Princeton University, 1982.

[10] H. Gelernter, "Realization of a Geometry-Theorem Proving Machine," in *Computers and Thought,* edited by E. A. Feigenbaum and J. Feldman, McGraw-Hill: New York, pp. 134-152, 1963.

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

[12] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation,* Addison-Wesley: Reading, 1979.

[13] I. Lakatos, *Proof and Refutations - The Logic of Mathematical Discovery* , edited by J. Worrall and E. Zahar, Cambridge University Press: Cambridge, 1976.

[14] D.B. Lenat, "AM: An Artificial Intelligence Approach to Discovery in Mathematics," Ph.D. dissertation, Department of Computer Science, Stanford University, 1976.

[15] Leyton, M. "A Process Grammar for Shape." *Artificial Intelligence* vol.34-2, pp. 213-247, 1988.

[16] Ore, O. *Theory of Graphs*, American Mathematical Society Colloquium Publications, Volume 38, American Mathematical Society: Providence, 1962.

[17] B. Pascal, *Pensées de Pascal*., Éditions Garnier Frères: Paris, pp. 73-84, 1964.

[18] G. Polya, *How to Solve It* , Doubleday Anchor Books: Garden City, 1957. 2nd ed.

[19] G. Polya, *Mathematical Discovery* , John Wiley & Sons: New York, 1962.Volumes 1 and 2.

[20] H. Poincaré, *La Valeur de la Scième*, Flammarion: France, pp. 27-40, 1970.

[21] T.W. Pratt and D.P. Friedman, "A Language Extension for Graph Processing and Its Formal Semantics," *CACM* vol. 14-7, pp. 460-467, 1971.

[22] A. Proskurowski, "Recursive Graphs, Recursive Labelings and Shortest Paths," *SIAM J. Comput.* vol. 10-2, pp. 391-397, 1981.

[23] B.K. Rosen, "Deriving Graphs from Graphs by Applying a Production," *Acta Informatica* vol. 4, pp. 337-357, 1975.

[24] M. Sims, "Empirical and Analytic Discovery in IL," in *Proceedings of the Fourth International Workshop on Machine Learning,* Irvine, CA, pp. 274-280, 1987.

[25] N.S. Sridharan,"Artificial Intelligence and Mathematical Reasoning," paper presented to the Mathematics Section meeting of the New York Academy of Sciences, New York, NY, May, 1984.

[26] P.E. Utgoff, "Shift of Bias for Inductive Concept Learning," in *Machine Learning: An Artificial Intelligence Approach*, Volume II, edited by R.S. Michalski, J.G. Carbonell and T.M. Mitchell, Tioga Publishing: Palo Alto, pp. 107-148, 1986.

[27] M. Wegman, "Summarizing Graphs by Regular Expressions," in POPL, pp. 203-216, 1983.

[28] M. Yannakakis, "Node-and-Edge-Deletion NP-Complete Problems," in *Proceedings 10th Annual ACM Symposium on Theory of Computing,* Association for Computing Machinery: New York, pp. 253-264, 1978.