GT: A Conjecture Generator for Graph Theory

Wing-Kwong Caeser Wong

AI TR86-21 January 1986

This work is sponsored in part by the Army Research Office under contract #DAAG29-84-K-0060 and NSF CER grant MCS8122039.
GT : A CONJECTURE GENERATOR FOR GRAPH THEORY

BY

WING-KWONG CAESAR WONG, B.S.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCES

THE UNIVERSITY OF TEXAS AT AUSTIN

December, 1985
ACKNOWLEDGEMENTS

I am very grateful to my advisors Dr. Elaine Rich and Dr. Bruce Porter. Dr. Rich encouraged me to work on this thesis initially and provided valuable advice and ideas. Dr. Porter gave me careful guidance throughout the project. And both advisors made very constructive comments on earlier drafts of this thesis that lead to significant improvement of its style and content. Through working with them, I have acquired valuable experience in doing research.

The Artificial Intelligence Lab of the Department of Computer Science provides excellent computing facilities and library resources, for which I am grateful.¹

I also thank several friends for their generous assistance. Rob Robinson and Braddlee helped to correct several drafts of this thesis. Chee often helped me when I had problems with the computing facilities.

I am greatly indebted to my parents, sisters, and brothers who persistently support me both spiritually and financially.

¹This work is sponsored in part by Army Research Office under contract DAAG29-84-K-0060 and NSF CER grant MCS8122039.
Finally, I whole-heartedly thank Huei-Ling for being my sincere, caring and inspiring companion for the last two years. I sincerely hope that we will continue to support and care for each other by sharing our lives together.

Wing-Kwong Caesar Wong

The University of Texas at Austin
December, 1985
ABSTRACT

The process of knowledge acquisition can be automated with programs that learn from observation and discovery. A better understanding of this learning strategy would facilitate the construction of expert systems and the exploration of scientific domains. GT, a frame-based program written for this thesis, learns relationships among classes of graphs by observing the examples and non-examples of these classes. The relationships considered are subclass, superclass, exclusion, and complement exclusion. GT's knowledge is partly represented as frames and partly as formulas of predicate calculus. The learned relationships is stored hierarchically. GT has found non-trivial, well-known theorems of graph theory.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgements</td>
<td>iv</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>vi</td>
</tr>
<tr>
<td>Chapter 1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1. Motivations for Learning Logical Relationships</td>
<td>2</td>
</tr>
<tr>
<td>1.2. A Brief Background of Machine Learning</td>
<td>5</td>
</tr>
<tr>
<td>1.2.1. A Unified View of Learning by Induction</td>
<td>6</td>
</tr>
<tr>
<td>1.2.2. Learning from Observation and Discovery</td>
<td>6</td>
</tr>
<tr>
<td>1.2.2.1. Meta-DENDRAL</td>
<td>7</td>
</tr>
<tr>
<td>1.2.2.2. BACON</td>
<td>8</td>
</tr>
<tr>
<td>1.2.2.3. AM\EURISKO</td>
<td>9</td>
</tr>
<tr>
<td>1.3. Organization of This Thesis</td>
<td>10</td>
</tr>
<tr>
<td>Chapter 2. Demonstration of the GT Program</td>
<td>12</td>
</tr>
<tr>
<td>2.1. Knowledge Representation</td>
<td>12</td>
</tr>
<tr>
<td>2.1.1. Graphs</td>
<td>13</td>
</tr>
<tr>
<td>2.1.1.1. VERTICES and NEXT-VS-SEQ</td>
<td>15</td>
</tr>
<tr>
<td>2.1.1.2. OUT-REACH-SEQ</td>
<td>16</td>
</tr>
<tr>
<td>2.1.1.3. OUT-UNIPATH-P-SEQ</td>
<td>16</td>
</tr>
<tr>
<td>2.1.2. Classes of Graphs</td>
<td>17</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.4.2. Verification of Conjectures</td>
<td>58</td>
</tr>
<tr>
<td>4.5. Conclusion</td>
<td>59</td>
</tr>
<tr>
<td>Appendix A. Glossary of Theory of Directed Graphs</td>
<td>61</td>
</tr>
<tr>
<td>Appendix B. Initial Graphs of GT</td>
<td>66</td>
</tr>
<tr>
<td>Appendix C. A Short Sample Run of GT</td>
<td>70</td>
</tr>
<tr>
<td>Appendix D. LISP Functions to Compute the OUT-REACH-SEQ of a Graph</td>
<td>76</td>
</tr>
<tr>
<td>Appendix E. More Results from the Basic Experiment</td>
<td>79</td>
</tr>
<tr>
<td>Bibliography</td>
<td>81</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 2-1: Four Sample Graphs in GT .......................... 14
Figure 2-2: Simplified Frames for the Graphs in Fig. 2-1 15
Figure 2-3: A Typical Frame of the Class ACYCLIC .... 17
Figure 2-4: Relationships Learned by GT in a Sample Run 22
Figure 3-1: The Graph Tri4b .................................... 29
Figure 3-2: The Transitive Closure of the Hierarchy of Figure 2-4 38
Figure 4-1: A NETWORK Graph Which is Not ACYCLIC 54
Figure 4-2: An ACYCLIC Graph and Its Perturbations 56
Chapter 1

Introduction

When a domain is being explored, many conceptual classes may look interesting. If positive and/or negative examples of these classes are available, one can learn the descriptions of these classes so that new instances can be classified. Some programs have been built to tackle this problem, e.g., [Winston 75] and [Mitchell 77].

In addition to class descriptions, relationships among classes are valuable information that should be learned. First, relationships can be used to infer the properties of one class from its related classes. Second, relationships pose empirical constraints. This helps the development of a theory that unifies the knowledge about the domain of the classes.

This thesis focuses on the problem of learning relationships:

GIVEN: (1) The characteristic descriptions of some classes of objects, and
(2) A number of objects to be classified

FIND: The relationships among those classes

Four logical relationships between any two classes are especially interesting: subclass, superclass, exclusion, and complement exclusion. One
class, say A, is a subclass of another, say B, or equivalently, B is a superclass of A, if all members of A belong to B (e.g. cats is a subclass of animals). One class excludes another if they have no common members (e.g. the classes of fishes and cats). Two classes are considered to be complement exclusions if their complements exclude one another (e.g. the classes of students whose GPA are above 2.0 and students whose GPA are below 3.0). Two classes are unrelated if none of the four relationships exists.

1.1 Motivations for Learning Logical Relationships

Many learning systems face the problems of forming meaningful concepts and making conjectures about their relationships. Some examples are AM [Lenat 76], CLUSTER [Michalski 83a], and JAUNDICE [Fu 85]. A new conceptual class is formed when similar objects are grouped together. For example, prime numbers form a meaningful class in AM since each prime has exactly two divisors.

The relationship between two classes is a general property that can be used as an inference rule. Let A and B be two classes. If A is a subclass of B, then any example of A will possess the properties of B. For example, knowing that cats are animals entails the expectation that a cat would have the power of locomotion, fixed structure and limited growth, and nonphotosynthetic metabolism, which are the characteristics of animals. For another example, if one believes that the class of polynomial problems and the class of non-polynomial problems do not overlap, then one would be able to infer there is little chance of finding a polynomial solution for a problem known to be non-polynomial.
Besides being used as inference rules, relationships also serve as causality indicators. A class of objects is generally characterized by a predicate which is a constraint on one or more descriptors of the objects in question. Therefore the relationship between two classes can be viewed as a relationship between the constraints of their characteristic predicates (e.g. multiples of six is a subclass of multiples of three which means that divisibility by six implies divisibility by three). This perspective is more explanatory because it shows some connection among the descriptors involved. Such a connection is sometimes suggested by observable data. Therefore, the discovery of some empirical relationship between two classes should inspire more in-depth investigation about the descriptors in question. A classic example in physical science is the formulation of the periodic law by the Russian chemist Mendeleyev in 1869. By observing that some elements exhibited similar chemical properties and then arranging all known elements in the order of their atomic weights, he discovered that these elements show a periodic variation in most of their properties. As a result, he classified the elements into mutually exclusive groups such as noble gases and halogens. This classification contributed to the development of atomic concepts by providing a fundamental datum which a satisfactory atomic theory must explain.

Besides serving as potential empirical laws, hypothetical relationships are sometimes universally true theorems. Another application of learning relationships arises in mechanical theorem proving. In formal domains such as mathematics, any hypothesis discovered by a learning program can be fed
into a theorem prover. If the hypothesis can be proved, then it is a valid theorem. Verification is desirable because theorems, but not hypotheses, can be used with complete confidence.

Since relationships among conceptual classes of any domain are practical and inspiring information about that domain, a learning program should empirically look for regularities and propose relationships among those classes whenever data are available. Most learning programs attempt to find out the relationships of subclass and superclass but ignore those of exclusion and complement exclusion. A program developed for this thesis, named GT (after Graph Theorist), is able to learn all four relationships among any given classes of graphs.

When given the characteristic descriptions of some classes of graphs, GT proposes some plausible relationships among those classes. If empirical evidence indicates that those classes are related, their relationships are constructed as some hierarchies of conjectures. When a new class is known, GT examines its relationships with the old classes and so may expand the hierarchies. From the hierarchies, GT is able to deduce all related classes of any given class.

In the following section, we will present a brief introduction to machine learning. This will provide some basic guidelines with which GT can be critically examined.
1.2 A Brief Background of Machine Learning

The ability to learn is one of the most precious and intelligent assets human beings possess. An intelligent machine should be able to acquire knowledge actively by observing, reasoning, and experimenting. Different views have been expressed on the meaning of learning, for example, [Simon 83] suggests that an agent learns to improve its performance on some specific task. For the purpose of this thesis, however, I consider learning as the formulation of interesting conjectures. In other words, an agent is learning if it is generalizing some specific information to form interesting rules.

Learning programs can be classified according to their underlying strategies, which fall into four general categories: rote-learning, learning by taking advice, by analogy, and by induction. [Michalski 83b], [Cohen 82], and [Rich 83] provide a comprehensive coverage of these principles and some representative programs that employ these principles.

Inductive learning can be sub-divided into two areas: concept learning from examples and learning by observation and discovery. This thesis falls into the latter area, about which more background material will be given. In particular, the next section gives a unified view of inductive learning while the section after summerizes three outstanding works of learning from observation and discovery.
1.2.1 A Unified View of Learning by Induction

[Simon 74] gives a unified view of the approach to inductive learning. Their model has two main components: the *instance space* and the *rule space*. The instance space is the set of all possible specific data points that can serve as input to the learning program. The rule space is the set of all possible rules or concepts that the program may learn. The learning problem is to find the rules that best describe the given training data.

Very often, the rules learned are not satisfactory since the given training instances are too limited or perhaps biased. Thus, the program should select its own training instances to refine the rules that are learned and to guide the search for new rules. The problem in this case is that of obtaining suitable data.

1.2.2 Learning from Observation and Discovery

In acquiring knowledge from observation and discovery, the learning element tries to obtain some general descriptions of a collection of observations. These descriptions can be heuristic rules, algorithms, empirical laws, or mathematical theorems.

Scientific discovery is a slow, expensive, and difficult process because it generally requires a lot of material resources, time, and human intelligence including knowledge and ingenuity. Programs that automate this process, even if partially, can be a great contribution.
In the subsections below, we will summarize three significant projects on learning from observation and discovery. They are distinguished by their domains, their methods of representing knowledge, and the contrasting methods they use to acquire various types of knowledge. From the experience of these programs, we can see what kinds of discovery is possible and what problems this type of learning faces.

1.2.2.1. Meta-DENDRAL

Meta-DENDRAL is a program that discovers cleavage rules when given different chemical structures and their mass spectra [Buchanan 78]. A cleavage rule for a particular structure predicts which bond(s) of the structure will break under unstable conditions. These rules are used by the DENDRAL system to indentify the molecular structure of unknown chemicals whose mass spectra are available.

Learning is achieved at first by model-directed generate-and-test search and then by fine hill-climbing in the rule space. The instance space is not searched because of the complexity involved. The program succeeds in discovering old well-known rules as well as new useful rules that were not reported before, despite the presence of noise in the given data.

Meta-DENTRAL is an important project because it works in a practical domain. It demonstrated the possibility of applications of artificial intelligence research as early as the mid-70's. Its major weakness is that much of its domain knowledge is buried in the code instead of being explicitly represented.
1.2.2.2. BACON

Instead of learning rules that can be used in an expert systems, BACON discovers empirical laws when given some noise-free numeric data of selected features [Langley 83]. A typical law that the program has learned is the ideal gas law which says $PV/nT$ is a constant, where $P$ is the pressure on the gas, $V$ is the volume, $n$ is the number of moles, and $T$ is the temperature.

The program has a small set of data-driven heuristics which detect regularities in the given numeric data. Some of the heuristics can define new theoretical terms such as the product and quotient of two features that fit certain trends. Regularities among basic features and theoretical terms are found when some new term has a constant value. It has re-discovered physical laws like Ohm’s law and Archimedes’ law of displacement, and chemical laws such as Proust’s law of definite proportions and Gay-Lussac’s law of combining volumes. Like Meta-DENDRAL, BACON does not search its instance space.

BACON is characterized by its small set of data-driven heuristics, which contrast sharply from the model-driven methods used by Meta-DENDRAL. Thus the BACON model can be used to work with areas whose domain theories are unknown. But BACON has two major weaknesses. First, it can be adversely affected by noisy data. Second, the relevant features it works on are pre-selected; otherwise, the program needs to work with both relevant and irrelevant features and combinatorial explosion will result.
1.2.2.3. AM\EURISKO

AM is a production system that discovers concepts and conjectures of set theory and elementary number theory by using many aesthetic-driven heuristics [Lenat 76]. It takes advantage of three powerful techniques: frame representation, production systems, and heuristically guided best-first search.

Frames are used to represent concepts. Properties of the concepts are stored in the slots of their frames. AM's major tasks include creating new frames and filling in the slots of all frames. These tasks are put on an agenda and will be executed in order, according to their priority scores.

Heuristics are used to generate interesting concepts and conjectures. Many of these heuristics are based on aesthetic considerations such as symmetry and extreme values. Some heuristics estimate the interestingness of each concept, which in turn affects the priority of various tasks to be executed. Thus it is a heuristically directed best-first search. Unlike MetaDENDRAL and BACON, AM searches both its rule space and instance space.

Some of AM's discoveries are natural numbers and primes, the unique-prime-factorization theorem, and Goldbach's conjecture. Despite these impressive results, AM has been further extended to a program called EURISKO [Lenat 83].

Unlike AM, EURISKO explicitly represents its heuristics as frames and so can manipulate them like frames of other concepts. As a consequence, new heuristics are generated and their interestingness estimated. This is an
improvement over having a fixed set of heuristics since the utilities of the old heuristics may decline as new situations emerge. EURISKO has claimed success in various domains such as LISP programming and VLSI design.

The AM/EURISKO models have demonstrated different techniques that are useful for making discoveries: the frame representation of concepts and heuristics, production systems, and heuristically directed best-first search. Their major limitations are that a good domain should have a high density of interesting concepts and that there exist simple heuristics to estimate the interestingness of these concepts.

Inspired by the success of the AM program, this thesis uses the techniques of frame representation and the induction principle to demonstrate how discoveries can be made in the domain of graph theory. The goal is to test the generality and utility of these techniques.

1.3 Organization of This Thesis

In Chapter two, we will demonstrate GT's performance with simple examples. This includes a short introduction to some concepts of directed graphs and their representation in GT, classification of graphs, some results of GT's learning, and organization of those results.

In Chapter three, a detailed description of the GT program will be given. We will examine several things. First, what concepts about directed graphs are used, and why. Second, what requirements the representation of the definitions of classes of graphs should satisfy and how the requirements
are satisfied. Third, how GT's learning is guided by three simple rules. Fourth, how GT organizes its knowledge about the relationships among any given classes, and why.

In Chapter four, the GT model will be critically evaluated. Its achievement, limitations and possible extensions will be discussed.
Chapter 2

Demonstration of the GT Program

This chapter presents some simple examples to demonstrate how and what GT learns. There are three main tasks in GT:

1. Classification of graphs.

2. Learning of the relationship between two classes by examining their positive and negative examples.

3. Organization of the relationships among all the classes in GT.

The first two tasks are illustrated by examining the representation of knowledge in GT in Section 1. In Section 2, we present examples of how GT organizes knowledge as it learns. The last section shows what GT has discovered in a typical run.

2.1 Knowledge Representation

GT has two main types of knowledge: Graphs and Classes of graphs. GT is initially given 29 graphs as a priori knowledge. When GT is invoked, the definitions of new classes are supplied by a user. GT then tests its initial graphs with respect to those classes and learns their relationships using their example and non-example graphs. In the next two subsections, we will show
how graphs and classes are represented in GT and how GT manipulates them to achieve learning.

2.1.1 Graphs

A graph is an abstraction of some structure and can be realized as a set of vertices and edges (ordered pairs of vertices.) The two vertices of an edge must be distinct. A graph in general can be directed or non-directed. All graphs are assumed to be directed in GT. Four of GT’s initial graphs are shown in figure 2-1. For example, TRI3B is a graph whose vertices are \{v1, v2, v3\} and whose edges are \{(v1 v2), (v1 v3), (v2 v3)\}. Notice that (v1 v2) is an edge leading from the vertex v1 to the vertex v2, (v1 v3) is an edge from v1 to v3, and (v2 v3) is an edge from v2 to v3.

In GT, each graph is represented as a frame. Simplified frames of the graphs in figure 2-1 are given in figure 2-2.

Actual frames for graphs in GT have more slots, a complete set of which will be given in the next chapter. GT has some procedures to compute the values of the properties such as OUT-REACH-SEQ and OUT-UNIPATH-P for any graph whose VERTICES and NEXT-VS-SEQ are given. Therefore, the user can always input any new graph to GT by supplying its VERTICES and NEXT-VS-SEQ. GT then computes the other properties and store them into a frame created for the new graph.

---

2 This restriction is an axiom in the traditional theory of directed graphs and can be dropped from here without affecting the principles of this thesis.
Figure 2-1: Four Sample Graphs in GT

The frames of graphs are used by GT to determine if each of those graphs belongs to a given class or not. In order to see how GT classifies graphs, it is necessary to understand the meaning of those slots.
(a) Graph Fourtoura
   VERTICES: (v1 v2 v3 v4)
   NEXT-VS-SEQ: ((v2 v3 v4) (v3 v4) (v4) nil)
   OUT-REACH-SEQ: ((v2 v3 v4) (v3 v4) (v4) nil)
   OUT-UNIPATH-P-SEQ: (nil nil t t)

(b) Graph Tri3b
   VERTICES: (v1 v2 v3)
   NEXT-VS-SEQ: ((v2 v3) (v3) nil)
   OUT-REACH-SEQ: ((v2 v3) (v3) nil)
   OUT-UNIPATH-P-SEQ: (nil t t)

(c) Graph Tri3d
   VERTICES: (v1 v2 v3)
   NEXT-VS: ((v2) (v1 v3) nil)
   OUT-REACH-SEQ: ((v1 v2 v3) (v1 v2) nil)
   OUT-UNIPATH-P-SEQ: (nil nil t)

(d) Graph Tri2c
   VERTICES: (v1 v2 v3)
   NEXT-VS-SEQ: ((v3) (v3) nil)
   OUT-REACH-SEQ: ((v3) (v3) nil)
   OUT-UNIPATH-P-SEQ: (t t t)

**Figure 2-2:** Simplified Frames for the Graphs in Fig. 2-1

### 2.1.1.1. VERTICES and NEXT-VS-SEQ

The VERTICES slot of a graph g represents the list of vertices of g. The NEXT-VS-SEQ slot represents the list of all (NEXT-VS v) corresponding to the order of vertex v in (VERTICES g), where (NEXT-VS v) is the list of all vertices u such that (v u) is an edge of g. For example, for the graph Tri3b in figure 2-1 (b), (NEXT-VS v1) = (v2 v3), (NEXT-VS v2) = (v3), (NEXT-VS v3) = nil. Hence, (NEXT-VS-SEQ Tri3b) = ((v2 v3) (v3) nil).
Note that the order of the elements in NEXT-VS-SEQ is important since its nth element refers to the NEXT-VS of the nth element of the VERTICES of the graph. Thus, VERTICES and NEXT-VS-SEQ uniquely determines a graph.

2.1.1.2. OUT-REACH-SEQ

The OUT-REACH-SEQ slot represents the list of all (OUT-REACH v) corresponding to the order of vertices v in (VERTICES g), where (OUT-REACH v) is the list of all vertices u that are reachable from v by following some edges.

\[ \text{e.g. (OUT-REACH-SEQ Tri3b) = ((v2 v3) (v3) nil) and (VERTICES Tri3b) = (v1 v2 v3)} \]
(Refer to figure 2-1 (c))

means (a) (OUT-REACH v1) = (v2 v3)
since v1 cannot reach itself and
v1 can reach v2 (via the path (v1 v2))
and v3 (via (v1 v2 v3)),
(b) (OUT-REACH v2) = (v3)
since v2 cannot reach v1 or itself,
and can reach v3 (via (v2 v3)), and
(c) (OUT-REACH v3) = nil
since v3 cannot reach any vertex.

2.1.1.3. OUT-UNIPATH-P-SEQ

The OUT-UNIPATH-P-SEQ slot represents the list of all (OUT-UNIPATH-P v) corresponding to the order of vertices v in (VERTICES g), where (OUT-UNIPATH-P v) is true if v cannot reach to itself and v can reach any vertex in (OUT-REACH v) via a unique path.
e.g. (OUT-UNIPATH-P-SEQ Tri3b) = (nil t t)  
(VERTICES Tri3b) = (v1 v2 v3) (refer to Fig. 2.1 (c))
means (a) (OUT-UNIPATH-P v1) is not true because v1
  can reach v3 via the paths (v1 v3) and (v1 v2 v3),
(b) (OUT-UNIPATH-P v2) is true because v2 can reach
  v3 via the only path (v2 v3)
  and v2 can reach neither v1 nor itself, and
(c) (OUT-UNIPATH-P v3) is true because v3 cannot reach
  any vertex.

2.1.2 Classes of Graphs

Frames are also used to represent classes of graphs. Each class-
description frame has the following slots: DEFINITION, EXAMPLES,
MISSES, SUBCLASSES, SUPERCLASSES, EXCLUSIONS, COMPLEMENT-
EXCLUSIONS (shortened as COMP-EXCLUS), and UNRELATED-CLASSES
(shortened as URCS). A typical frame of the class ACYCLIC is shown in
figure 2-3.

DEFINITION: (ALL v VERTICES
  (NOT (MEMBER v (OUT-REACH v))))
EXAMPLES: (FOURTOURA TRI2C TRI3B)
MISSES: (TRI3D)
SUBCLASSES: (FOREST)
SUPERCLASSES: NIL
EXCLUSIONS: NIL
COMPLEMENT-EXCLUSIONS: NIL
UNRELATED-CLASSES: NIL

Figure 2-3: A Typical Frame of the Class ACYCLIC

After the definition of a new class is input to GT by the user, GT
tests its initial graphs to determine if they satisfy the definition. The graph's name will be inserted into the EXAMPLES slot or the MISSES slot of the frame of the class, depending on whether it satisfies the definition. Then, using the classified graphs, GT learns the relationships among all classes by filling in the slots of SUBCLASSES, SUPERCLASSES, EXCLUSIONS, COMP-EXCLUS, and URCS (collectively referred to as the relationship slots) of those classes.

2.1.2.1. DEFINITION

A new class is formed in GT when the user inputs the definition of that class. A graph belongs to the class if and only if it satisfies the class's definition. A lisp-like predicate calculus is used to represent each definition.

Let us look at two examples. The definition of the class ACYCLIC is:

\[(\text{ALL } v \ \text{VERTICES} \ (\text{NOT} \ (\text{MEMBER} \ v \ (\text{OUT-REACH} \ v))))\]

which says for any vertex \(v\) of the graph in question, \(v\) does not belong to the list \((\text{OUT-REACH} \ v)\), i.e., \(v\) cannot reach itself through any cycle. Notice that NOT and MEMBER are LISP functions.

The definition of the class FOREST is:

\[(\text{ALL } v \ \text{VERTICES} \ (\text{OUT-UNIPATH-P} \ v))\]

which says for any vertex \(v\), \(v\) cannot reach itself and \(v\) can reach any vertex of \((\text{OUT-REACH} \ v)\) in a unique path, according to the definition of OUT-UNIPATH-P.
2.1.2.2. EXAMPLES and MISSES

GT tests its initial graphs against the definition of a new class and stores the examples in the class's EXAMPLES slot and the non-examples in the MISSES slot.

Suppose GT had the initial graphs shown in figure 2-1 and 2-2. Let us see what the result of classifying those graphs with respect to ACYCLIC and FOREST would be:

(EXAMPLES ACYCLIC) = (Fourtoura Tri2c Tri3b)
(MISSES ACYCLIC) = (Tri3d)

Remember a graph is ACYCLIC if each of its vertices, say v, does not belong to (OUT-REACH v). On checking the slots VERTICES and OUT-REACH-SEQ of the frames of the graphs Fourtours, Tri2c, and Tri3b, GT finds that all of them satisfy this requirement. Tri3d is a non-example of ACYCLIC because GT notices that the vertex v1 of Tri3d is a member of (OUT-REACH v1).

(EXAMPLES FOREST) = (Tri2c)
(MISSES FOREST) = (Fourtoura Tri3b Tri3d)

Remember a graph is a FOREST if the OUT-UNIPATH-P of each of its vertices is true. By examining the slots VERTICES and OUT-UNIPATH-P of the frames of the graphs, GT determines that Tri2c is the only graph in figure 2-1 that satisfies this condition.
2.1.2.3. Relationship Slots

GT's major learning task is to fill in the relationship slots of all classes. After GT has classified the initial graphs with respect to all new classes, GT compares pairs of classes one at a time. After examining the examples of misses of the two classes in question, GT fills in some of the relationship slots of both classes.

For example, upon finding the EXAMPLES slot of ACYCLIC, whose value is (FOURTOURA TRI2C TRI3B), contains all graphs in the same slot of FOREST, whose value is (TRI2C), GT would propose that ACYCLIC is a superclass of FOREST by inserting the name FOREST into the SUBCLASSES slot of ACYCLIC and the name ACYCLIC into the SUPERCLASSES slot of FOREST.

2.2 Organization of the Learned Relationships

GT is capable of learning the relationships among any number of classes. If there are many classes to be related, a large number of relationships will result. If GT does not organize these relationships in a compact way, the result can be very confusing to the user. Because of this consideration, GT stores a minimum number of relationships in a hierarchical way.

For example, if GT had learned that ROOTED-TREE is a subclass of FOREST and FOREST is a subclass of ACYCLIC, then GT can infer that ROOTED-TREE is subclass of ACYCLIC and would not bother to empirically check the relationships of these two classes. So, GT would not store the redundant relationship between ROOTED-TREE and ACYCLIC.
In another case, suppose GT had learned that ROOTED-TREE is a subclass of ACYCLIC and that ROOTED-TREE is a subclass of FOREST. If now GT learns that FOREST is a subclass of ACYCLIC, then the relationship between ROOTED-TREE and ACYCLIC is redundant and would be removed from the stored knowledge of GT.

2.3 Result of a Sample Run of GT

In a single run, GT is given the definitions of 0-1-IN-DEG, ACYCLIC, BINARY-TREE, FOREST, NETWORK, and ROOTED-TREE. GT first obtains the examples and non-examples of these classes using the initial graphs. After all classes have been related, GT has discovered several conjectures which are shown as a hierarchy in figure 2-4. The exact run and the meanings of these classes are given in the appendix.

The result in figure 2-4 can be re-stated as the following conjectures (the meanings of special terms about graphs can be found in the appendix):

1. A BINARY-TREE is a ROOTED-TREE, i.e.,
   (AND (ALL V VERTICES
           (AND (OUT-UNIPATH-P V)
                (OR (EQ (OUT-DEG V) 0) (EQ OUT-DEG V) 2)))
           (UNIQUE V VERTICES
               (AND (NOT (MEMBER V (OUT-REACH V)))
                    (EQ (OUT-REACH-SIZE V) (SUB1 NUM-VS)))))
   implies
   (AND (ALL V VERTICES (OUT-UNIPATH-P V))
        (UNIQUE V VERTICES
            (AND (NOT (MEMBER V (OUT-REACH V)))
                 (EQ (OUT-REACH-SIZE V) (SUB1 NUM-VS))))).
(a) Hierarchy of Subclass Relationships

- 0-1-IN-DEG
- FOREST
- ROOTED-TREE
- BINARY-TREE
- ACYCLIC
- NETWORK

(b) Exclusions: BINARY-TREE and NETWORK

Figure 2-4: Relationships Learned by GT in a Sample Run

2. Each vertex of a ROOTED-TREE has at most one in-coming edge, i.e.,
   (AND (ALL V VERTICES (OUT-UNIPATH-P V))
    (UNIQUE V VERTICES
     (AND (NOT (MEMBER V (OUT-REACH V)))
      (EQ (OUT-REACH-SIZE V) (SUB1 NUM-VS)))))

implies
   (ALL V VERTICES (OR (EQ (IN-DEG V) 0)
                      (EQ (IN-DEG V) 1))).

This is a well-known theorem of graph theory.
( [Robinson 80] p.116 Theorem 5.3.)
3. A ROOTED-TREE is a FOREST, i.e.,
\[(\text{AND} (\text{ALL V VERTICES} (\text{OUT-UNIPATH-P V}))
\quad \text{(UNIQUE V VERTICES)}
\quad \text{(AND} (\text{NOT} (\text{MEMBER V (OUT-REACH V))})
\quad \text{(EQ} (\text{OUT-REACH-SIZE V}) (\text{SUB1 NUM-VS}))))\)
\implies
\text{(ALL V VERTICES} (\text{OUT-UNIPATH-P V})).

4. A FOREST is ACYCLIC, i.e.,
\[(\text{ALL V VERTICES} (\text{OUT-UNIPATH-P V}))
\implies
\text{(ALL V VERTICES} (\text{NOT} (\text{MEMBER V (OUT-REACH V}))))).

5. A NETWORK is ACYCLIC, i.e.,
\[(\text{AND} (\text{UNIQUE V VERTICES})
\quad \text{(AND} (\text{NOT} (\text{MEMBER V (OUT-REACH V))})
\quad \text{(EQ} (\text{OUT-REACH-SIZE V}) (\text{SUB1 NUM-VS}))))
\quad \text{(UNIQUE V VERTICES})
\quad \text{(AND} (\text{NOT} (\text{MEMBER V (IN-REACH V))})
\quad \text{(EQ} (\text{IN-REACH-SIZE V}) (\text{SUB1 NUM-VS}))))\)
\implies
\text{(ALL V VERTICES} (\text{NOT} (\text{MEMBER V (OUT-REACH V}))))).

6. A graph cannot be a NETWORK and a BINARY-TREE at the same time, i.e.,
\[(\text{AND} (\text{UNIQUE V VERTICES})
\quad \text{(AND} (\text{NOT} (\text{MEMBER V (OUT-REACH V))})
\quad \text{(EQ} (\text{OUT-REACH-SIZE V}) (\text{SUB1 NUM-VS}))))
\quad \text{(UNIQUE V VERTICES})
\quad \text{(AND} (\text{NOT} (\text{MEMBER V (IN-REACH V))})
\quad \text{(EQ} (\text{IN-REACH-SIZE V}) (\text{SUB1 NUM-VS}))))\)
\implies
\text{(NOT}
\quad \text{(AND} (\text{ALL V VERTICES})
\quad \text{(AND} (\text{OUT-UNIPATH-P V})
\quad \text{(OR} (\text{EQ} (\text{OUT-DEG V}) \text{ 0})
\quad \text{(EQ} \text{ OUT-DEG V}) \text{ 2])))
\quad \text{(UNIQUE V VERTICES})
\quad \text{(AND} (\text{NOT} (\text{MEMBER V (OUT-REACH V))})
\quad \text{(EQ} (\text{OUT-REACH-SIZE V})
\quad \text{(SUB1 NUM-VS}))))))].
Notice that GT does not store redundant conjectures such as 'A ROOTED-TREE is ACYCLIC' since this conjecture can be inferred from the third and fourth conjectures given above. Notice also that all the above conjectures, except the fifth one (see figure 4-1), are valid theorems. This indicates that GT is capable of making decent discoveries as well as false conjectures.
Chapter 3

Anatomy of GT

This chapter will present a detailed description of the GT program. In section 1, we will present an overview of GT. Section 2 discusses the representation of graphs and classes of graphs and how graphs are classified. Section 3 discusses the heuristic rules for learning relationships. Section 4 examines the way GT organizes the conjectured relationships. This chapter elaborates on Chapter 2 by stressing the motivations, heuristics, analysis, and algorithms of how GT learns.

3.1 Overview of GT

GT consists of a user-interface, a classifier, and a relationship sponsor which is the major learning element. The user-interface allows the user to interact with GT to input descriptions of classes of graphs and new sample graphs. The classifier decides if a given graph belongs to a given class. The relationship sponsor compares the positive and negative examples of two selected classes and suggests their relationship.

A high level description of the flow of control in GT is:
DO-UNTIL user satisfied
Get new class from user
Obtain examples and/or non-examples of the new class using GT’s initial graphs
Choose a pair of classes
DO-UNTIL all classes are related
  IF the two classes are not already related THEN
    Learn relationship between the pair of classes
    Remove all redundant relationships suggested previously
  FI
  Choose the next pair of classes
OD
OD

3.2 Knowledge Representation

Graphs and classes of graphs are the two major types of knowledge in GT. In order to manipulate these entities, GT explicitly represents them as frames, each of which has a fixed number of slots. Each slot stands for an attribute of the entity represented. There are two types of frame operations: filling in slots and reading from them. When all slots of a frame are filled, it contains information about many aspects of the entity represented. (See [Minsky 75] and [Rich 83] for more discussion on knowledge representation by frames.)

Two graphs should have the same set of attributes so their frames use the same slots although the values of each slot of the graphs may be different. On the other hand, the attributes of graphs and classes are not the same, so the slots of their frames are different.
In section 1 of chapter two, the representation of graphs and classes are illustrated with examples. Here, we will focus on the motivations and advantages of the representation.

3.2.1 Properties of Graphs

The interpretation of each slot of a graph frame is shown below:

VERTICES: List of the vertices of the graph

NUM-VS: Number of vertices

NUM-ES: Number of edges

NEXT-VS-SEQ: List of (NEXT-VS v) for all vertices v corresponding to the order of v in VERTICES, where (NEXT-VS v) is the destination vertices of all out-going edges of v

OUT-DEGREE-SEQ:
List of (OUT-DEGREE v) which is the number of vertices in (NEXT-VS v)

OUT-REACH-SEQ:
List of (OUT-REACH v) which is the vertices reachable from v

OUT-REACH-SIZE-SEQ:
List of (OUT-REACH-SIZE v) which is the number of vertices in (OUT-REACH v)

OUT-MAXMIN-SEQ:
List of (OUT-MAXMIN v) which is the maximum length of a path from v to any vertex of (OUT-REACH v) if v has no loop, or otherwise is the minimum length of a loop of v
OUT-UNIPATH-P-SEQ:
List of (OUT-UNIPATH-P v) which is true iff v has no loop and v cannot reach any vertex in more than one path

MIN-LOOP-SEQ: List of (MIN-LOOP v) which is the minimum length of a loop of v, or NUM-VS plus one if v has no loop

PRE-VS-SEQ: List of (PRE-VS v) which is the beginning vertices of all in-coming edges of v

IN-DEGREE-SEQ:
List of (IN-DEGREE v) which is the number of vertices in (PRE-VS v)

IN-REACH-SEQ: List of (IN-REACH v) which is the vertices from which v is reachable

IN-REACH-SIZE-SEQ:
List of (IN-REACH-SIZE v) which is the number of vertices in (IN-REACH v)

The meanings of special terms about graphs (e.g. destination vertices, out-going edges, loop, and length of a path) can be found in the glossary in the appendix. Four slots, VERTICES, NEXT-VS-SEQ, OUT-REACH-SEQ, and OUT-UNIPATH-P-SEQ, were illustrated in section 2.1.1. The graph Tri4b and its frame are shown in figure 3-1. Notice that VERTICES and NEXT-VS-SEQ are necessary to specify a graph uniquely; other properties can be derived from them.
(a) Diagram of Tri4b

(b) The Frame of Tri4b

VERTICES: (v1 v2 v3)
NUM-VS: 3
NUM-ES: 4
NEXT-VS-SEQ: ((v2 v3) (v3) (v2))
OUT-DEGREE-SEQ: (2 1 1)
OUT-REACH-SEQ: ((v2 v3) (v2 v3) (v2 v3))
OUT-REACH-SIZE-SEQ: (2 2 2)
OUT-MAXMIN-SEQ: (1 2 2)
OUT-UNIPATH-P-SEQ: (nil nil nil)
MIN-LOOP-SEQ: (4 2 2)
PRE-VS-SEQ: (nil (v1 v3) (v1 v2))
IN-DEGREE-SEQ: (0 2 2)
IN-REACH-SEQ: (nil (v1 v2 v3) (v1 v2 v3))
IN-REACH-SIZE-SEQ: (0 3 3)

Figure 3-1: The Graph Tri4b
3.2.1.1. Why Graph Properties Are Needed

Why does GT need all these properties of a graph? The major reason is that this allows many interesting classes of graphs to be concisely described. For example, the concepts of *cycles*, *sink*, and *source* help to define the classes of ACYCLIC and NETWORK. A graph is ACYCLIC if each of its vertices does not lie on a cycle; a graph is a NETWORK if the graph has a unique source and a unique sink.

A vertex $v$ which lies on some cycle means $v$ can reach itself via a sequence of edges. A vertex $v_1$ can reach some other vertex $v_2$ if there exists a path leading from $v_1$ to $v_2$. A path is a sequence of vertices, any consecutive pair of which is an edge. For example, in the graph Tri4b, $(v_1 \ v_3 \ v_2)$ is the path consisting of the edges $(v_1 \ v_3)$ and $(v_3 \ v_2)$ in this order. So, we define the property OUT-REACH of a vertex $v$ as all vertices that $v$ can reach. Thus, a vertex $v$ which lies on a cycle can be described as $v$ is a member of (OUT-REACH $v$). Now, it is easy to define the class ACYCLIC: a graph is ACYCLIC if for any vertex $v$ of the graph, $v$ is not a member of (OUT-REACH $v$).

Consideration of other interesting classes shows that graph properties such as OUT-REACH-SEQ would help to define succinctly many of those classes. For example, the class of STRONG graphs (more commonly known as strongly connected graphs) can be characterized by the simple condition: For any $v_1$, $v_2$ which are vertices of the graph in question, $v_1$ belongs to (OUT-REACH $v_2$).
3.2.1.2. Frames as Storage Structure

GT has some procedures to compute the values of the properties of a graph given its VERTICES and OUT-REACH-SEQ. The user can always input new graphs as wanted. It is important to note that these procedures are simply LISP functions instead of being explicitly represented in the GT program. This is so because GT does not need to manipulate or do any reasoning with them. The LISP functions to compute OUT-REACH-SEQ from VERTICES and NEXT-VS-SEQ of a graph are given in the appendix. They show the formal definition of OUT-REACH-SEQ of a graph.

It should be pointed out that the frames of graphs are just static objects serving as data. This means they are never modified once they are created. Then why does GT store the values of every property into frame slots instead of computing them when needed? The major reason is that a graph will be classified under each class in GT. In order to avoid redundant computation of the same properties of each graph, it is more time-efficient to store them. The trade-off is that more space is used to store the extra slots of the frames.

3.2.2 Definitions of Classes

A class is also represented as a frame. Its slots include DEFINITION, EXAMPLES, MISSES, SUBCLASSES, SUPERCLASSES, EXCLUSIONS, COMPLEMENT-EXCLUSIONS, and NON-RELATED-CLASSES. (AM uses similar frames, with a lot more slots, to represent its concepts [Lenat 76]) These slots are demonstrated with examples in chapter 2 on page 17. Among
these slots, DEFINITION is the most critical, since a class is distinguished from others by its definition. So, we consider further the representation of definitions of classes in this section.

There are three basic requirements on the representation of the DEFINITION of a class:

1. Understandability: Since a class is created in GT when a user inputs its definition, the representation should be concise and readily understandable to the user. This motivates the creation of various properties of a graph, as discussed in section 3.2.1.1.

2. Generality: The representation allows the descriptions of many interesting classes such as ACYCLIC, FOREST, STRONG, NETWORK, etc.

3. Decidability: The representation allows GT to decide if a graph belongs to a given class. Using the classified graphs, GT would then learn the relationships among all classes.

Since many common classes require all vertices or a unique vertex to satisfy some constraint, predicate calculus is a useful representation for definitions of classes. A typical predicate has the form:

(Quantifier variable scope-of-the-variable constraint)

Remember a graph is ACYCLIC if each of its vertices does not lie on any cycle. Its formal definition in GT is as concise as:

\[
(\text{ALL } v \text{ VERTICES } (\text{NOT } (\text{MEMBER } v (\text{OUT-REACH } v))))
\]

The class of ROOTED-TREE can be characterised as 'the graph in
question must be a FOREST, and there is a unique vertex which can reach
every other vertex of the graph but not itself,’ which can be formalized as:

\[
(\text{AND} (\text{ALL} \ v \ \text{VERTICES} \ \text{(OUT-UNIPATH-P} \ v)) \\
(\text{UNIQUE} \ v \ \text{VERTICES} \\
(\text{AND} (\text{NOT} \ (\text{MEMBER} \ v \ (\text{OUT-REACH} \ v))) \\
(\text{EQ} \ (\text{OUT-REACH-SIZE} \ v) \ (\text{SUB1} \ \text{NUM-VS}))))))
\]

Notice that (1) \((\text{ALL} \ v \ \text{VERTICES} \ \text{(OUT-UNIPATH-P} \ v))\) is the definition of
FOREST, (2) \((\text{OUT-REACH-SIZE} \ v)\) is the number of vertices in \((\text{OUT-REACH} \ v)\), and (3) \(\text{NUM-VS}\) is the number of vertices of the graph.

Remember also STRONG graphs can be characterized by the
condition ‘for any \(v_1, v_2\) which are vertices of the graph in question, \(v_1\) can
reach \(v_2\),’ which can be nicely formalized in GT as:

\[
(\text{ALL} \ v_1 \ \text{VERTICES} \\
(\text{ALL} \ v_2 \ \text{VERTICES} \ (\text{MEMBER} \ v_2 \ (\text{OUT-REACH} \ v_1))))
\]

The properties of a vertex that are defined for GT are: NEXT-VS,
OUT-DEGREE, OUT-REACH, OUT-REACH-SIZE, OUT-MAXMIN, OUT-
UNIPATH-P, MIN-LOOP, PRE-VS, IN-DEGREE, IN-REACH, and IN-
REACH-SIZE. Different classes can then be defined in GT’s formalism by
using these properties, any LISP functions such as \text{NOT} and \text{MEMBER}, and
GT’s special terms such as the quantifiers \text{ALL} and \text{UNIQUE}. This formalism
is so general that many interesting classes of graphs can be represented. Also,
the syntax of the representation is concise and understandable. Therefore, the
LISP-like predicate calculus used in GT satisfies the requirements of
generality and understandability.
In order to decide if a graph belongs to a certain class, GT checks to see if the graph satisfies the definition of that class. For example, to decide if a graph is STRONG, GT will consider all possible pairs of vertices, say v1 and v2, of the graph and test if v2 is a member of (OUT-REACH v1). This includes checking the OUT-REACH-SEQ and VERTICES slots of the frame of the graph in question and executing the LISP function MEMBER. Since each of these tests only look at a finite number of things, the representation of the definitions of classes thus satisfies the requirement of decidability.

3.3 Learning Relationships

As pointed out in chapter 1, there are four common logical relationships between any two classes: subclass, superclass, exclusion, and complement exclusion. Notice that the first two can occur for a pair of classes at the same time. In this case, the two classes are referred as being equivalent, e.g., the class of multiples of six and the class of numbers that are divisible by both two and three. Similarly, the relationships of exclusion and complement exclusion can simultaneously exist between two classes, which are then complementary. For example, prime numbers and composite numbers are complementary.

The major task of GT is to learn the relationships among any number of given classes of graphs. Such tasks distinguish GT from most other learning programs (e.g. AM [Lenat 76], AQ-11 [Michalski 78], CLUSTER [Michalski 83a], and JAUNDICE [Fu 85].) First, GT assumes the definitions of the classes while the other programs try to learn those
definitions. Second, GT learns the relationships between all possible pairs of classes while the other programs do not. For example, AM does not relate prime numbers to numbers with perfect squares.

An inductive approach can be employed to make conjectures about concepts that are associated with empirical data. Suppose a sociology student is studying a group of American cities and is interested in two particular features, namely education budget and crime rate of a city. If he notices that most of the cities with a relatively small budget for education have high crime rates and yet many other cities swamped with crime have relatively large budgets for education, he would suspect that lack of education expenditure somehow contributes to crime rate but not the reverse.

GT works in a similar but more formal way. There are three heuristic rules, the first of which is also used by AM, to learn the relationship between any two classes Class1 and Class2 (assume the same set of instances have been classified with respect to both classes):

(H1) If all known examples of Class1 are also known examples of Class2, then induce Class1 is a subclass of Class2.

(H2) If the known examples of Class1 and those of Class2 do not overlap, then induce the two classes are exclusions.

(H3) If the known misses of Class1 and those of Class2 do not overlap, then induce the two classes are complement exclusions.

After all classes are instantiated with examples and/or non-examples,
GT learns the relationship between two chosen classes, say C1 and C2, with the following procedure:

Try heuristic H1 to see if C1 is a subclass of C2;
Try the same heuristic to see if C2 is a subclass of C1 (since C1 and C2 may be equivalent);
IF both trials fail, THEN
Try heuristic H2;
Try heuristic H3;
(since subclass or superclass relationship imply the impossibility of exclusions and complement exclusions)
FI

If the condition of a rule is not satisfied by two classes, then the relationship suggested in the action-part of the rule does not exist. Otherwise, the relationship is learned by GT. Hence, if the heuristic rules are tried but none of them succeeds, then the classes C1 and C2 are unrelated.

It is important to note that if any non-trivial intrinsic relationship indeed exists between C1 and C2, the relationship will always be detected by these heuristic rules because there exists no empirical data to contradict such relationship. This characteristic is the major thrust of these heuristics. Moreover, these general heuristics are independent of the domain of graph theory and are applicable under many different situations.
3.4 Organization of Learned Relationships

In order to use or communicate the knowledge that a system has learned, the system needs some way to organize the knowledge. In the case of GT, the major knowledge is the relationships among all the classes given. The problem here is to decide what learned relationships should be stored so that (1) GT can find all the classes that are related to a given class, and (2) the user can understand the relationships clearly.

Suppose there are n classes, the number of possible pairs of classes is $n^* (n-1)/2$. This is the maximum number of relationships that GT can learn. GT can choose to empirically detect and store the relationship between every possible pair of classes, but the resulting organization of knowledge would be confusing to the user because there are too many relationships stored. For the sake of understandability, GT only stores the necessary relationships that allow one to infer the relationship between every possible pair of classes. This is like representing the relationships as a hierarchy instead of as a transitive closure, which has edges between a vertex and all of its OUT-REACH of the original hierarchy. (Compare figures 2-4 and 3-2).

Now we should distinguish the explicit relationships from the implicit ones. The former relationships are stored in GT while the latter are not stored but can be inferred from the former group. An explicit relationship in the example in figure 2-4 is 'ROOTED-TREE is a subclass of FOREST' while an implicit relationship is 'ROOTED-TREE is a subclass of ACYCLIC'.
GT maintains its knowledge of relationships as a hierarchy in two ways: (a) Avoid redundant empirical comparison of two classes whose relationship is already stored or can be inferred. (b) Remove redundantly stored relationships after the relationship between two classes is learned.

3.4.1 Avoidance of Redundant Learning

Suppose GT has already learned that Class1 is a subclass of Class2 and that Class2 is a subclass of Class3. Now GT faces the task of relating Class1 and Class3. Before applying the heuristic rules in section 3.3 to check their relationship, GT would find out they are already related implicitly (Class1 is a subclass of Class3) by using the current knowledge of GT. So GT would skip the task of comparing the examples and non-examples of both classes.
To find out if two classes are already related with its current knowledge, GT uses several deductive inference rules. In the following rules, f, g, and h symbolize any three distinct classes.

(DR1) (((f is a subclass of g) and
    (g is a subclass of h))
  or
  ((f and g are exclusions) and
    (g and h are complement exclusions))) implies
  (f is a subclass of h)

(DR2) (((f is a subclass of g) and
    (g and h are exclusions)) implies
  (f and h are exclusions)

(DR3) (((f and g are complement exclusions) and
    (g is a subclass of h)) implies
  (f and h are complement exclusions)

(DR4) (((f and g are equivalent) or
    (f and g are complementary)) and
  (f and h are unrelated)) implies
  (g and h unrelated)


If GT were to learn relationships without first checking whether they already exist implicitly in the knowledge base, then GT’s learned relationships would result as a confusing hierarchy closure rather than a clear hierarchy. Therefore, even though extra time is required to deduce if two classes are implicitly related, I believe this is justified because simplicity and understandability of the organization of the knowledge are more important here.
3.4.2 Removal of Redundant Relationships

Although GT would avoid empirically detecting the relationship between two classes that are implicitly related, it could happen that some stored relationships become redundant after a new relationship is suggested by one of the heuristic rules. Suppose GT has learned that Class1 is a subclass of Class2 and that Class1 and Class3 are exclusions. Then if GT finds that Class2 and Class3 are exclusions, the explicit relationship between Class1 and Class3 becomes redundant since it can be inferred with the deductive rule DR2. By analyzing the rules DR1 to DR4, one can tell all the cases when an explicit relationship become redundant and should be removed. All these cases are considered by GT.

Suppose h, r, and s are distinct classes. There are three major cases when redundant explicit relationships should be removed depending on the new relationship empirically found between r and s:
(RR1) r is a subclass of s \[\text{[rel1]},\]

For each concept h such that
r is an explicit subclass of h \[\text{[rel2]},\] and
s is a (explicit or implicit) subclass of h \[\text{[rel3]},\]
Remove \[\text{[rel2]}\]
since \[\text{[rel1]}\] and \[\text{[rel3]}\] imply \[\text{[rel2]}\] with Rule DR1;

For each concept h such that
r and h are explicit exclusions \[\text{[rel4]},\] and
s and h are exclusions \[\text{[rel5]},\]
Remove \[\text{[rel4]}\]
since \[\text{[rel5]}\] and \[\text{[rel1]}\] imply \[\text{[rel4]}\] with Rule DR2;

For each concept h such that
h is an explicit subclass of s \[\text{[rel6]},\] and
h is a subclass of r \[\text{[rel7]},\]
Remove \[\text{[rel6]}\] from S
since \[\text{[rel1]}\] and \[\text{[rel7]}\] imply \[\text{[rel6]}\] with Rule DR1;

For each concept h such that
s and h are explicit complement-exclusions \[\text{[rel8]},\]
and
r and h are complement-exclusions \[\text{[rel9]},\]
Remove \[\text{[rel8]}\]
since \[\text{[rel9]}\] and \[\text{[rel1]}\] imply \[\text{[rel8]}\] with Rule DR3.
(RR2) r and s are exclusions \[ \text{[rel1]} \]

For each concept h such that
r is an explicit subclass of h \[ \text{[rel2]} \], and
s and h are complement exclusions \[ \text{[rel3]} \],
Remove \[ \text{[rel2]} \]
since \[ \text{[rel1]} \] and \[ \text{[rel3]} \] imply \[ \text{[rel2]} \] with Rule DR1;

For each concept h such that
h and r are explicit exclusions \[ \text{[rel4]} \], and
h is a subclass of s \[ \text{[rel5]} \],
Remove \[ \text{[rel4]} \]
since \[ \text{[rel5]} \] and \[ \text{[rel1]} \] imply \[ \text{[rel4]} \] with Rule DR2

For each concept h such that
h and s are explicit exclusions \[ \text{[rel6]} \], and
h is a subclass of r \[ \text{[rel7]} \],
Remove \[ \text{[rel6]} \]
since \[ \text{[rel7]} \] and \[ \text{[rel1]} \] imply \[ \text{[rel6]} \] with Rule DR2;

For each concept h such that
s is an explicit subclass of h \[ \text{[rel8]} \], and
r and h are complement exclusions \[ \text{[rel9]} \],
Remove \[ \text{[rel8]} \]
since \[ \text{[rel1]} \] and \[ \text{[rel9]} \] imply \[ \text{[rel8]} \] with Rule DR1.
(RR3) r and s are complement exclusions \([\text{rel1}]\)

For each concept \(h\) such that
- \(r\) and \(h\) are explicit complement exclusions \([\text{rel2}]\),
- and
- \(s\) is a subclass of \(h\) \([\text{rel3}]\),
Remove \([\text{rel2}]\)
since \([\text{rel1}]\) and \([\text{rel3}]\) imply \([\text{rel2}]\) with Rule DR3;

For each concept \(h\) such that
- \(h\) is an explicit subclass of \(r\) \([\text{rel4}]\), and
- \(h\) and \(s\) are exclusions \([\text{rel5}]\),
Remove \([\text{rel4}]\)
since \([\text{rel5}]\) and \([\text{rel1}]\) imply \([\text{rel4}]\) with Rule DR1;

For each concept \(h\) such that
- \(h\) is an explicit subclass of \(s\) \([\text{rel6}]\), and
- \(h\) and \(r\) are exclusions \([\text{rel7}]\),
Remove \([\text{rel6}]\)
since \([\text{rel7}]\) and \([\text{rel1}]\) imply \([\text{rel6}]\) with Rule DR1;

For each concept \(h\) such that
- \(s\) and \(h\) are explicit complement exclusions \([\text{rel8}]\), and
- \(r\) is a subclass of \(h\) \([\text{rel9}]\),
Remove \([\text{rel8}]\)
since \([\text{rel1}]\) and \([\text{rel9}]\) imply \([\text{rel8}]\) with Rule DR3.

The algorithms RR1 to RR3 are independent of the domain of graph theory and so are useful to any system that wants to remove redundant relationships in order to maintain the knowledge base of the system as a hierarchy.
Chapter 4

Evaluation of GT

This chapter begins with an example that successfully uses GT to make discoveries. Then several reasons which explain GT’s success and indicate GT’s limitations will be discussed. As a result, some solutions are suggested to combat these limitations. Finally, conclusions are presented.

4.1 Basic Experiment

The definitions of some interesting classes of graphs fit the same syntactic pattern. For example, the following definitions all fit the pattern (ALL V VERTICES (NOT (MEMBER V (<vertex property> V)))):

- (ALL V VERTICES (NOT (MEMBER V (OUT-REACH V)))), which designates the class of all acyclic graphs.

- (ALL V VERTICES (NOT (MEMBER V (IN-REACH V)))), which designates the class of all acyclic graphs.

- (ALL V VERTICES (NOT (MEMBER V (NEXT-VS V)))), which designates the class of all possible graphs.

- (ALL V VERTICES (NOT (MEMBER V (PRE-VS V)))), which designates the class of all possible graphs.
Another interesting pattern is \((\text{ALL} \ V \ (\text{EQ} \ (<\text{vertex property}> \ V) \ <\text{maximum value of the vertex property}>))\), which covers the following classes:

- Strong graphs, which is designated by the definition \((\text{ALL} \ V \ (\text{EQ} \ (\text{OUT-REACH-SIZE} \ V) \ \text{NUM-VS})))\).

- Strong graphs, which is designated by the definition \((\text{ALL} \ V \ (\text{EQ} \ (\text{IN-REACH-SIZE} \ V) \ \text{NUM-VS})))\).

- Complete graphs, which is designated by the definition \((\text{ALL} \ V \ (\text{EQ} \ (\text{OUT-DEGREE} \ V) \ (\text{SUB1} \ \text{NUM-VS})))\).

- Complete graphs, which is designated by definition \((\text{ALL} \ V \ (\text{EQ} \ (\text{IN-DEGREE} \ V) \ (\text{SUB1} \ \text{NUM-VS})))\).

- Acyclic graphs, which is designated by the definition \((\text{ALL} \ V \ (\text{EQ} \ (\text{MIN-LOOP} \ V) \ (\text{ADD1} \ \text{NUM-VS})))\).

So I have chosen quite a few special patterns to run an experiment with GT. By substituting various vertex properties into these patterns, many class definitions are generated and given to GT, which then proceeds to learn the relationships among these classes. Some other interesting classes like STRONG-TOURNAMENT are also given to GT.

The chosen patterns are as follows:

Let \(<\text{VP}>\) stands for an arbitrary vertex property.

(P1) If \(<\text{VP}>\) is a list of vertices, e.g. NEXT-VS, OUT-REACH,
(a) \((\text{ALL} \ V \ \text{VERTICES} \ (\text{MEMBER} \ V \ (<\text{VP}> \ V)))\)
(b) \((\text{ALL} \ V \ \text{VERTICES} \ (\text{NOT} \ (\text{MEMBER} \ V \ (<\text{VP}> \ V)))\)
(c) \((\text{ALL} \ V1 \ \text{VERTICES}
\quad (\text{ALL} \ V2 \ \text{VERTICES} \ (\text{MEMBER} \ V2 \ (<\text{VP}> \ V1))))\)
(P2) If <VP> is an integer, e.g., OUT-DEGREE, MIN-LOOP,
   (note: the range of the value of each numeric vertex
    property is given to GT initially)
   (a) (ALL V VERTICES
       (EQ (<VP> V) <maximum of the range of <VP>>))
   (b) (ALL V VERTICES
       (EQ (<VP> V)
        (SUB1 <maximum of the range of <VP>>)))
   (c) (ALL V VERTICES
       (EQ (<VP> V)
        (ADD1 <minimum of the range of <VP>>)))
   (d) (ALL V VERTICES
       (EQ (<VP> V) <minimum of the range of <VP>>))
   (e) (ALL V1 VERTICES
       (ALL V2 VERTICES
        (NOT (EQ (<VP> V1) (<VP> V2))))))
   (f) (a) with ALL replaced by UNIQUE
   (g) (b) with ALL replaced by UNIQUE
   (h) (c) with ALL replaced by UNIQUE
   (i) (d) with ALL replaced by UNIQUE
   (j) (EQ <sum of values of <VP> of each vertex> NUM-ES)
   (k) (EQ <sum of values of <VP> of each vertex>
       (SUB1 NUM-ES))

(P3) If <VP> is boolean, e.g., OUT-UNIPATH-P,
   (a) (ALL V VERTICES (<VP> V))
   (b) (ALL V VERTICES (NOT (<VP> V)))))
   (c) (a) with ALL replaced by UNIQUE in (a)
   (d) (b) with ALL replaced by UNIQUE in (b)

Many interesting results are discovered by GT. Some of them are:

- Each vertex of a STRONG-TOURNAMENT has a MIN-LOOP value of 3. ( [Robinson 80] p. 102, Theorem 4.13.) This is a most surprising result.

- (ALL V VERTICES (NOT (MEMBER V (OUT-REACH V)))
  (ALL V VERTICES (NOT (MEMBER V (IN-REACH V))))), and
(ALL V VERTICES (EQ (MIN-LOOP V) (ADD1 NUM-VS))) all refer to the class of acyclic graphs.

- A graph is STRONG iff OUT-REACH-SIZE of each of its vertices equals the number of vertices of the graph.

- A graph is STRONG iff IN-REACH-SIZE of each vertex equals the number of vertices of the graph.

- (ALL V VERTICES (EQ (OUT-DEGREE V) (SUB1 NUM-VS))) and (ALL V VERTICES (EQ (IN-DEGREE V) (SUB1 NUM-VS))) both refer to the class of complete graphs.

- Each vertex of a STRONG graph belongs to its IN-REACH.

- Each vertex of a graph belongs to its IN-REACH iff each vertex belongs to its OUT-REACH.

- A STRONG graph must not be ACYCLIC.

Meanings of the classes mentioned above and more conjectures that GT has learned can be found in the appendix.

Also, four classes, collectively referred to as universal classes, have been discovered to be extreme. Their definitions are:

1. The sum of IN-DEGREE's of all vertices equals the number of edges of a graph, i.e., (EQ (SUM IN-DEGREE-SEQ) NUM-ES).

2. Each vertex v does not belong to (NEXT-VS v), i.e., (ALL V VERTICES (NOT (MEMBER V (NEXT-VS V)))�)

3. The sum of OUT-DEGREE's of all vertices equals the number of edges of a graph, i.e., (EQ (SUM OUT-DEGREE-SEQ) NUM-ES).

4. Each vertex v does not belong to (PRE-VS v), i.e., (ALL V VERTICES (NOT (MEMBER V (PRE-VS V)))).
Since all graphs in GT belong to each of the universal classes, GT conjectures that every given class is a subclass of each of these universal classes. In fact, every possible class of graphs will be a subclass of a universal class because the definition of a universal class represents an intrinsic property of graphs.

4.2 Why GT Appears to Work

Under the double-space view of [Simon 74] given in Chapter 1 on page 6, GT performs a special case of learning by observation because of two reasons. First, GT's rule space is limited to just four logical relationships. Previous works usually use heuristics to guide the search in a huge rule space. Second, the instance space is not searched since all graphs are given to GT. Thus GT does not attempt to test its conjectures.

Despite GT's simplicity, some of the conjectures learned by GT are quite impressive. They include familiar theorems like 'Each vertex of a rooted tree has at most one in-coming edge' and 'Each vertex of a strong tournament has a loop of length three.' It is important to understand why GT can achieve such discoveries. We point out several reasons below:

1. Some meaningful characteristics of the objects in question can be identified and given to GT a priori, e.g., OUT-REACH and OUT-UNIPATH-P of a vertex of a graph.

2. Interesting classes can be formed by constraining some of those characteristics. For example, a graph is ACYCLIC (which is a common class of graphs) if each of its vertices, say v, does not belong to (OUT-REACH v).
3. The definitions of all classes are given to GT by the user.

4. Some of the classes given to GT are in fact related in one of the four known relationships: subclass, superclass, exclusions, and complement exclusions (e.g., FOREST is a subclass of ACYCLIC.) Otherwise, GT would not discover any genuine relationships.

5. The conjectures to be learned can be represented in a concise form. For example, the simple act of inserting the name FOREST in the SUBCLASSES slot of the frame of ACYCLIC is asserting the rather complicated theorem '(ALL v VERTICES (OUT-UNIPATH-P v)) implies (ALL v VERTICES (NOT (MEMBER v (OUT-REACH v))))'.

6. Sufficiently large set of reasonably well selected instances are given a priori. This reduces the number of false conjectures made by GT.

7. The learning heuristics used can always detect genuine intrinsic relationships. But at the same time, they can be oversensitive, i.e., they can generate many wrong conjectures, if insufficient or biased data are used.

Several of the above points are actually restrictions of the GT model in that major declarative and procedural knowledge is either programmed into GT (e.g. procedures that obtain the values of the properties of each graph), or given to GT by the user (e.g. class definitions and sample graphs). In order to improve the generality of the GT model, these restrictions should be removed. In the following two sections, we will discuss two general problems that cover these restrictions and suggest ways to solve them.
4.3 Problem of a Fixed Set of Concepts

A major deficiency of GT is obvious when it is compared with other programs that learn by induction: GT lacks the ability to create new properties of graphs and new classes of graphs. New concepts are important because they are necessary for the development of a deeper and wider understanding of the domain in question. In number theory, the discovery of the concept of prime numbers is valuable to the understanding of numbers. From the concept of prime number, more interesting concepts are discovered in AM and so the knowledge about the domain is expanded [Lenat 76].

4.3.1 Suggested Solutions

In AM, new concepts are generated by mutating the definitions of old concepts by using heuristics. These concepts include classes of objects (such as numbers) and operations (such as set union). Every concept is a frame, which includes slots such as domain and range if the frame represents an operation.

GT has operations, which are LISP functions, to compute the values of various properties of a graph given its VERTICES and NEXT-VERTICES. But GT cannot manipulate these functions to get new operations like AM does. In order to achieve this, GT needs to represent the operations explicitly, perhaps as frames. Also, heuristics are needed by GT to guide its search of new interesting concepts. Sample heuristics can be found in [Lenat 76] and [Michalski 83c].
In addition to discovering new graph properties, GT can also be modified to generate new classes. One simple experiment that we have done with GT is to let it create a new class when two classes are found to be unrelated. This new class is the intersection of these two classes, i.e., its definition is the conjunction of the definitions of both classes. This allows GT to create concepts like ROOTED-TREE (from FOREST and SINGLE-SOURCE), BINARY-TREE (from ROOTED-TREE and 0-2-OUT-DEG), and STRONG-TOURNAMENT (from STRONG and TOURNAMENT), which are quite interesting classes. So it seems some simple heuristics can improve GT's learning capability.

Although AM's method of learning by heuristic search produces impressive results, the method has its limitations. The result of running AM shows a lot of interesting discoveries made within a certain period, but its performance declines rapidly after that, although there is still a lot to be explored about number theory. This is due to the fact that AM lacks the ability to create new heuristics [Lenat 83], so Lenat built another program called EURISKO that represents heuristics explicitly as frames which can be manipulated by the program. As a result, new heuristics can be created in the same way as new classes and operations. EURISKO is claimed to be successful in different domains such as LISP programming and VLSI design. So it suggests that EURISKO's method of learning heuristics can also be used to improve the capability of GT.
4.3.2 Further Discussion

I have spent a fair amount of time to extend GT based on the AM model but most of the efforts failed. There are several points I think are worth noting:

- In GT, most of the procedures to compute graph properties are quite complicated. The procedures involve many simpler concepts (see Appendix D for examples). AM's model requires the procedures to be represented explicitly as frames so that GT can (1) execute their definitions to obtain the values of the properties of given graphs, and (2) mutate their definitions to produce new interesting properties. These constraints entail that all the simpler concepts also need to be represented as frames. Since there are a lot of such concepts, encoding them as frames would take a lot of time. Moreover, there are other serious problems, to be described below, about the AM model. I have therefore chosen not to finish this encoding task.

- Since the procedures of computing graph properties are complicated, it is hard to find general heuristics to mutate the definitions of these procedures to generate new meaningful properties. The chance of success is very small.

- The problem of generating new classes is not as difficult as the above two problems since class definitions are relatively simple. There are many ways with which to obtain new classes. For example, a new class can be formed when a relationship is contradicted by a few instances. But then the problem is how to measure the interestingness of a class so that only interesting classes can be focused on.

- The interestingness of a graph, a class of graphs, and properties of graphs are hard to measure. There are few known criteria and heuristics that are both general and useful.

From the above discussion, it should be clear that several principles of the AM model are not suitable for GT. The reasons can be summarized as:
(1) AM starts with simple concepts and discovers new concepts of moderate complexity, (2) GT starts with complex concepts, and (3) the problem of generating complex concepts from complex concepts is much harder than the problem of discovering complex concepts from simple ones.

4.4 Problem of Induction

Empirical data can be used to induce hypotheses that assert universal truths (e.g., that all heavenly bodies are spherical) but no data can establish those hypotheses with complete confidence. This is because a hypothesis generally covers an infinite number of instances (e.g., contexts of different places and times) while only a finite number of instances can be used to induce this hypothesis in reality. This is a fundamental problem of induction in the enterprise of science [Popper 65].

For the above reason, GT sometimes make false conjectures. This problem can be reduced if more and better selected data are available. This is known as the problem of searching the instance space. Suppose GT is given the concept of tertiary tree, i.e., a rooted-tree whose vertices have three or no out-going edges. But GT's initial graphs do not include any tertiary tree, so GT would wrongly learn that the class of tertiary trees is a subclass of every given class. If GT can generate or is given examples of tertiary trees, then GT would not make such mistake.

Another general case may happen. Say GT makes the conjecture that Class1 is a subclass of Class2 because the known examples of Class1 are
also known to belong to Class 2. This conjecture may be wrong since there may exist graphs that belong to Class 1 but not Class 2 and that are not known to GT. For example, NETWORK is mistaken as a subclass of ACYCLIC by GT because among all the graphs in GT, all NETWORK graphs are ACYCLIC (see Chapter 2 on page 21.) This is wrong because there are graphs that are NETWORK but not ACYCLIC. An example is given in figure 4-1. Remember a graph is ACYCLIC if each of its vertices does not lie on any circle and a NETWORK is a graph with a unique source and a unique sink.

![Diagram](image)

**Figure 4-1:** A NETWORK Graph Which is Not ACYCLIC

Two general principles should be applied to reduce the induction problem. (1) More examples should be generated automatically to test the conjectures suggested. (2) Attempts should be made to verify the conjectures.
4.4.1 Falsification of Conjectures

Although the fundamental problem of induction cannot be solved, the problem can be reduced by testing the induced hypotheses with more empirical data. A philosophy of science proposed by Popper requires a hypothesis be examined critically before it is accepted [Popper 65]. Since a graph is a finite formal object, GT can randomly generate graphs to test every conjecture it makes. But many classes have relatively small sizes and random graphs would normally be negative examples of those classes. Thus this method is too weak to test the relationships learned by GT.

The idea of perturbation can be used to guide the search of the instance space. For example, in PET, which is a program that learns to solve symbolic integration and linear equations by experimental goal regression, new sample problems can be generated by applying some perturbation operators to a problem supplied by the user. These new problems are then used to refine the rules PET has learned [Porter 85].

The idea of perturbation operators is applicable in GT. Since a graph is a set of vertices and a set of edges, one can form a new graph by making small changes to the vertices and edges of a given graph. I believe four simple operators are especially useful in GT:

(PO1) Add an edge.
(PO2) Delete an edge.
(PO3) Reverse an edge.
(PO4) Add a vertex and an edge connected to this vertex.

Some examples can demonstrate the utility of these operators.
Figure 4-2: An ACYCLIC Graph and Its Perturbations

Given an example of ACYCLIC graph, which is shown in figure 4-2(A), how would one apply the perturbation operators to get new examples of ACYCLIC?
Case 1: By PO1. Two vertices, say $x$ and $y$, are chosen. If (C1) $x$ and $y$ are distinct, (C2) $(x \ y)$ is not an edge in the original graph, and (C3) $x$ does not belong to (OUT-REACH $y$), then a new ACYCLIC graph can be obtained by adding the edge $(x \ y)$ to the old graph. For example, a new ACYCLIC graph is obtained by adding the edge $(v1 \ v3)$ to Four4b (see figure 4-2(B) (a) and (b)).

Case 2: By PO2. Deleting an edge of an ACYCLIC graph would always produce another ACYCLIC graph (see figure 4-2(B)(c)).

Case 3: By PO3. Two vertices, say $x$ and $y$, are chosen. If (1) $x$ and $y$ are distinct, (2) $(x \ y)$ is an edge in the old graph, and (3) $x$ cannot reach $y$ via other path, then replacing the edge $(x \ y)$ by the edge $(y \ x)$ would always produce another ACYCLIC graph (see figure 4-2(B) (d) and (e)).

Case 4: By PO4. Adding a vertex and an edge connected to this vertex to an ACYCLIC graph would always produce another ACYCLIC graph (see figure 4-2(B)(f)).

From the examples, we can see that applications of the perturbation operators to an example of a class are more likely to produce new examples than would random generation of graphs without the initial example and the operators. Also, perturbation can be guided by constraints such as C1, C2, and C3 in case 1 above. If these constraints can be learned in the process of generating new examples, the perturbation method can be further improved. PET attempts to solve this problem by using relational models [Porter 85].
There are other general techniques to guide the search of the instance space. For example, AM generates examples for a concept by executing the procedural specification of the concept with its known examples [Lenat 76]. Another example is Kim's EGS program which does automatic example generation using a transformational approach [Kim 85]. Both techniques can be used by GT to generate more graphs to test its conjectures.

4.4.2 Verification of Conjectures

In addition to falsifying conjectures, GT's results can become more reliable if the results can be verified. This can be attempted by using some mechanical theorem provers such as Boyer and Moore Theorem Prover [Boyer 79].

Since a mechanical prover requires theorems and axioms to be stated in a representation that they can work on, the knowledge about the properties like OUT-REACH-SEQ and OUT-UNIPATH-P in GT must be represented explicitly if theorems about these properties are to be proved. For example, the Boyer and Moore prover works on theorems written in a logic similar to pure LISP.

Although automatic theorem proving is a very nice feature to have in the process of discovering theorems, it has several limitations. First, many theorems are too complicated to be provable by any available mechanical theorem prover. Second, some theorem provers, like the Boyer and Moore prover, need human assistance to supply lemmas before a theorem can be
proved. In addition, in domains where knowledge is not complete or probabilistic events are involved, verification is out of the question. Examples of such knowledge are empirical laws and rules for medical diagnosis.

I have tried to prove, with Boyer and Moore logic, some theorems discovered by GT. In most cases, a lot of simpler functions need to be defined and a lot of lemmas need to be supplied before the theorems can be proved. This indicates that many theorems discovered by GT are quite complex so proving them is a hard problem.

4.5 Conclusion

Finding the relationships among conceptual classes of objects is important since (1) We can use relationships as inference rules to classify new objects. (2) These relationships serve as interesting conjectures one can investigate further; new theories are often formulated with respect to constraints derived from empirical data. A model of learning relationships by empirical evidence is discussed and implemented by the program GT in order to work with directed graphs.

GT uses frames to represent graphs and classes of graphs. Class definitions are represented by a LISP-like predicate calculus.

GT uses three general heuristics to induce the relationships between two classes that have some known positive and/or negative examples. The major power of these heuristics lies in the fact that no genuine intrinsic relationship will be empirically contradicted. Their weakness is that they can be oversensitive when poor data are used.
GT maintains the simplicity and understandability of the learned relationships in two ways. First, it avoids redundant learning of relationships that already exist implicitly in the knowledge base. Second, it removes from the knowledge base redundant relationships that can be inferred from other stored relationships. The resulting knowledge of GT will be in a compact hierarchical form. This organization is independent of the domain of graph theory.

GT is a special case of learning from observation. Its rule space is restricted to four logical relationships and it does not search its instance space. Despite its simplicity, GT has discovered non-trivial theorems found in text books of graph theory. This shows that GT's simple inductive method of learning relationships is quite useful.

Nevertheless, GT has two major limitations: (1) The definitions of classes and the properties of graphs are given. (2) The conjectures made by GT are not tested automatically. With these limitations, GT can at most be a tool to assist mathematicians in exploring the domain of graph theory. In order to promote GT as a more conscientious researcher, these assumptions must be removed. In particular, we have discussed learning by heuristic search and indicated how it may be used in GT. This would allow GT to generate new concepts by heuristic search. We have also suggested a method of generating new example graphs by using perturbation operators. These graphs should be used to test the conjectures GT has made.
Appendix A.

Glossary of Theory of Directed Graphs

0-1-IN-DEG: A class of graphs g such that vertices of g have at most one in-coming edge, i.e.,

\[ \text{(ALL V VERTICES} \]  
\[ \text{(OR (EQ (IN-DEGREE V) 1))} \]  
\[ \text{(EQ (IN-DEGREE V) 0))}) \]

0-2-OUT-DEG: A class of graphs g such that vertices of g have either two or no out-going edges, i.e.,

\[ \text{(ALL V VERTICES} \]  
\[ \text{(OR (EQ (OUT-DEGREE V) 2))} \]  
\[ \text{(EQ (OUT-DEGREE V) 0))}) \]

ACYCLIC: A class of graphs g such that no vertex of g is reachable to itself, i.e.,

\[ \text{(ALL V VERTICES} \]  
\[ \text{(NOT (MEMBER V (OUT-REACH V)))}} \]

BINARY-TREE: A class of graphs g such that g is a ROOTED-TREE and g has vertices which have either two or no out-going edges, i.e.,

\[ \text{(AND (definition of ROOTED-TREE)} \]  
\[ \text{(definition of 0-2-OUT-DEG))} \]

COMPLETE: A class of graphs g such that for any two vertices v1 and v2 of g, the edge (v1 v2) always exists, i.e.,

\[ \text{(ALL V1 VERTICES} \]  
\[ \text{(ALL V2 VERTICES} \]  
\[ \text{(MEMBER V2 (NEXT-VS V1)))}} \]
Cycle of a graph: A path whose first and last vertices coincide.

Edge of a graph g:
A pair of vertices (v1 v2) where v1 is the beginning and v2 the destination of the edge.

FOREST: A class of graphs g such that each vertex v of g has no loop and v is reachable to any of its OUT-REACH in one and only one path, i.e.,

\[(\text{ALL } v \text{ VERTICES (OUT-UNIPATH-P } v))\]

Graph: A set of vertices and edges. A graph is also uniquely defined by a list of vertices and a list of NEXT-VS's corresponding to the order of the vertices.

IN-DEGREE of a vertex v:
The cardinality of (PRE-VS v).

IN-REACH of a vertex v:
All vertices that are reachable to v.

IN-REACH-SIZE of a vertex v:
The cardinality of (IN-REACH v).

In-coming edge of a vertex v:
An edge whose destination is v.

MIN-LOOP of a vertex v:
If v lies on some cycle, then MIN-LOOP is the minimum length of such cycles. Otherwise, MIN-LOOP is the number of edges of g plus one.

Length of a cycle: The number of unique vertices in the cycle.

Length of a path/walk:
The number of vertices in the path/walk.

Loop of a vertex v:
A cycle that includes v.
NETWORK: A class of graphs $g$ such that $g$ has a unique source and a unique sink, i.e.,

\[(\text{AND})\]

\[(\text{UNIQUE V VERTICES})\]

\[(\text{AND})\] \((\text{NOT (MEMBER V (OUT-REACH V)))}\)

\[(\text{EQ})\] \((\text{OUT-REACH-SIZE V})\)

\[(\text{SUB1 NUM-VS}))\]

\[(\text{UNIQUE V VERTICES})\]

\[(\text{AND})\] \((\text{NOT (MEMBER V (IN-REACH V)))}\)

\[(\text{EQ})\] \((\text{IN-REACH-SIZE V})\)

\[(\text{SUB1 NUM-VS}))\]

NEXT-VS of a vertex $v$:
The vertices each of which is the destination of an out-going edge of $v$.

OUT-DEGREE of a vertex $v$:
The cardinality of (NEXT-VS $v$).

Out-going edge of a vertex $v$:
An edge whose beginning is $v$.

OUT-REACH of a vertex $v$:
All vertices that are reachable from $v$.

OUT-REACH-SIZE of a vertex $v$:
The cardinality of (OUT-REACH $v$).

OUT-MAXMIN of a vertex $v$:
The maximum length of a path from $v$ to any vertex of (OUT-REACH $v$) if $v$ has no loop, or the minimum length of a cycle including $v$ otherwise.

Path of a graph $g$:
A walk whose vertices are distinct.

PRE-VS of a vertex $v$:
The vertices each of which is the beginning of an in-coming edge of $v$. 
Reachability from a vertex v1 to a vertex v2:
The existence of a walk from v1 to v2.

ROOTED-TREE: A class of graphs g such that g is a FOREST and has a
unique source, i.e.,
(AND <definition of FOREST>
 (UNIQUE V VERTICES
  (AND (NOT (MEMBER V (OUT-REACH V)))
   (EQ (OUT-REACH-SIZE V)
   (SUB1 NUM-VS)))))

Sink of a graph g: A vertex which is reachable from all other vertices of g but
not itself.

Source of a graph g:
A vertex which is reachable to all other vertices of g but
not itself.

STRONG: A class of graphs g such that for any vertices v1 and v2 of
g, v1 can reach v2, i.e.,
(ALL V1 VERTICES
 (ALL V2 VERTICES
  (MEMBER V2 (OUT-REACH V1)))))

TOURNAMENT: A class of graphs g such that for any distinct vertices v1
and v2 of g, either the edge (v1 v2) or the edge (v2 v1) exists, i.e.,
(ALL V1 VERTICES
 (ALL V2 VERTICES
  (XOR (MEMBER V2 (NEXT-VS V1))
   (MEMBER V1 (NEXT-VS V2))))

Note that XOR is a special predicate defined in GT such
that (XOR p1 p2) is true if either p1 is true or p2 is true
but not both.

OUT-UNIPATH-P of a vertex v:
A predicate that returns true iff v has no loop and v is not
reachable to any vertex in more than one path.
Vertex of a graph $g$:
   A basic component of $g$.

Walk of a graph $g$:
   A list of vertices from which any pair of consecutive elements must be an edge of $g$. 
Appendix B.

Initial Graphs of GT

5tours

5treea

cycle4

four4a

fourall

fourtours
Appendix C.

A Short Sample Run of GT

** The sample graphs GT has are: (stoura streea cycle4 four4a fourall fourtoura fourtourb fourtourc train4 tree7a tri0 tri1 tri2a tri2b tri2c tri2d tri3a tri3b tri3c tri3d tri4a tri4b tri4c tri4d tri5 tri6 two0 two1 two2)

**** Do you want to create a new class of graphs and relate them?

Yes

**** Please type in 'quit' or the name of a new class and then its definition

0-1-IN-DEG

(ALL V VERTICES (OR (EQ (IN-DEG V) 0)
  (EQ (IN-DEG V) 1)) )

** Examples of 0-1-IN-DEG are: (streea cycle4 four4a train4 tree7a tri0 tri1 tri2a tri2b tri2d tri3a tri3d two0 two1 two2)
Misses are: (stoura fourall fourtoura fourtourb fourtourc tri2c tri3b tri3c tri4a tri4b tri4c tri4d tri5 tri6)

**** Please type in 'quit' or the name of a new class and then its definition
ACYCLIC

(ALL V VERTICES (NOT (MEMBER V (OUT-REACH V))))

** Examples of ACYCLIC are: (5treea fourtoura train4
  tree7a tri0 tri1 tri2a tri2b tri2c tri3b two0 two1)
Misses are: ...
(* GT's graphs minus the above examples *)

**** Please type in 'quit' or the name of a new class
and then its definition

BINARY-TREE

(AND (ALL V VERTICES (AND (UNIPATH-P V)
            (OR (EQ (OUT-DEG V) 0)
                (EQ (OUT-DEG V) 2))
        (UNIQUE V VERTICES
            (AND (NOT (MEMBER V (OUT-REACH V)))
                (EQ (OUT-REACH-SIZE V) (SUB1 NUM-VS))))))

** Examples of BINARY-TREE are: (5treea tree7a tri2a)
Misses are: ...

**** Please type in 'quit' or the name of a new class
and then its definition

FOREST

(ALL V VERTICES (UNIPATH-P V))

** Examples of FOREST are: (5treea train4 tree7a tri0
  tri1 tri2a tri2b tri2c two0 two1)
Misses are: ...

**** Please type in 'quit' or the name of a new class
and then its definition
NETWORK

(AND (UNIQUE V VERTICES
       (AND (NOT (MEMBER V (OUT-REACH V)))
            (EQ (OUT-REACH-SIZE V) (SUB1 NUM-VS))))
    (UNIQUE V VERTICES
       (AND (NOT (MEMBER V (IN-REACH V)))
            (EQ (IN-REACH-SIZE V) (SUB1 NUM-VS)))))

** Examples of NETWORK are: (fourtoura train4 tri2b tri3b two1)
Misses are: ...

**** Please type in 'quit' or the name of a new class
and then its definition

ROOTED-TREE

(AND (ALL V VERTICES (UNIPATH-P V))
    (UNIQUE V VERTICES
       (AND (NOT (MEMBER V (OUT-REACH V)))
            (EQ (OUT-REACH-SIZE V) (SUB1 NUM-VS)))))

** Examples of ROOTED-TREE are: (5treea train4 tree7a tri2a tri2b two1)
Misses are: ...

**** Please type in 'quit' or the name of a new class
and then its definition

quit

** Now, creating tasks to relate the new classes just created

****** Task 1 — (Relate NETWORK ROOTED-TREE)

** There is no logical relationship between the two classes
(*) because their examples overlap and their misses
overlap *)

...

(* tasks which do not relate classes are not shown in
order to put more focus on GT's major tasks *)

****** Task 3 -- (Relate 0-1-IN-DEG ROOTED-TREE)

** 0-1-IN-DEG looks like a superclass of ROOTED-TREE
(* because the examples of 0-1-IN-DEG include all
the examples of ROOTED-TREE *)

****** Task 4 -- (Relate 0-1-IN-DEG NETWORK)

** There is no logical relationship between the two
classes

...

****** Task 6 -- (Relate ACYCLIC 0-1-IN-DEG)

** There is no logical relationship between the two
classes

****** Task 7 -- (Relate ACYCLIC NETWORK)

** ACYCLIC looks like a superclass of NETWORK

...

****** Task 9 -- (Relate BINARY-TREE ACYCLIC)

** BINARY-TREE looks like a subclass of ACYCLIC
(* because the examples of BINARY-TREE are also
examples of ACYCLIC *)

****** Task 10 -- (Relate BINARY-TREE 0-1-IN-DEG)

** BINARY-TREE looks like a subclass of 0-1-IN-DEG
***** Task 12 — (Relate FOREST 0-1-IN-DEG)

** There is no logical relationship between the two classes

***** Task 13 — (Relate ROOTED-TREE BINARY-TREE)

** ROOTED-TREE looks like a superclass of BINARY-TREE

** The conjectured relationship of superclass between 0-1-IN-DEG and BINARY-TREE is removed
(* because this relationship can be inferred from that between 0-1-IN-DEG & ROOTED-TREE and that between ROOTED-TREE & BINARY-TREE *)

***** Task 14 — (Relate NETWORK BINARY-TREE)

** NETWORK and BINARY-TREE look like exclusions

***** Task 15 — (Relate BINARY-TREE FOREST)

** BINARY-TREE looks like a subclass of FOREST

***** Task 16 — (Relate ROOTED-TREE ACYCLIC)

** ROOTED-TREE looks like a subclass of ACYCLIC

** The conjectured relationship of superclass between ACYCLIC and BINARY-TREE is removed
(* because this relationship can be inferred *)

***** Task 17 — (Relate ACYCLIC ROOTED-TREE)

** The relationship between these two classes is already learned to be superclass
(* GT check if two classes are related by the existing conjectures before empirically detecting the relationship *)
****** Task 18 — (Relate ROOTED-TREE FOREST)

** ROOTED-TREE looks like a subclass of FOREST

** The conjectured relationship of subclass between BINARY-TREE and FOREST is removed
(* because this relationship can be inferred *)

...

****** Task 20 — (Relate NETWORK FOREST)

** There is no logical relationship between the two classes

...

****** Task 22 — (Relate ACYCLIC FOREST)

** ACYCLIC looks like a superclass of FOREST

** The conjectured relationship of superclass between ACYCLIC and ROOTED-TREE is removed
(* because this relationship can be inferred *)

...
Appendix D.

LISP Functions to Compute the OUT-REACH-SEQ of a Graph

(* The function OUT-REACH-SEQ returns a list of OUT-REACH of the vertices of a graph g corresponding to the order of the vertices in VS. VS is the list of G's remaining vertices whose OUT-REACH are to be computed one at a time. *)

(OUT-REACH-SEQ VS G) =
(COND ((LISTP VS) (CONS (OUT-REACH (CAR VS) G) (OUT-REACH-SEQ (CDR VS) G)))
        (T NIL))
(* The function OUT-REACH
returns the OUT-REACH of a
vertex V of a graph G. The OUT-REACH of V is
a list of vertices that can be reached
by V. The VERTICES and NEXT-VS-SEQ of G are
stored as properties of G. The NEXT-VS-SEQ of G
is a list of the NEXT-VS of each vertex of G.
The NEXT-VS of V is a list of destination
vertices of the out-going edges from V. *)

(OUT-REACH V G)
=
(0-REACH (NEXT-VS-OF-V V
  (GETPROP G 'VERTICES)
  (GETPROP G 'NEXT-VS-SEQ))
NIL
  (GETPROP G 'VERTICES)
  (GETPROP G 'NEXT-VS-SEQ))

(* The function 0-REACH
returns all the vertices that can be
reached by the vertices in KIDS. If each vertex of
KIDS is already known to be
reachable, then no new vertices that are reachable
can be found. Otherwise, for those vertices in KIDS
that are newly known to be reachable, include their
NEXT-VS's as reachable vertices and call 0-REACH
recursively. *)

(0-REACH KIDS REACHED VERTICES NEXT-VS-SEQ)
=
(COND ((SUBSETP KIDS REACHED) REACHED)
  (T (0-REACH (NEXT-VS-OF-V
    (SET-MINUS KIDS REACHED)
    VERTICES NEXT-VS-SEQ)
    (UNION REACHED KIDS)
    VERTICES
    NEXT-VS-SEQ))))
(* The function NEXT-VS-OF-V  
   returns the NEXT-VS of vertex V.  
   (NTH n y) returns the nth element of the list y.  
   (LOCATE x y) returns the numeric position of the  
   element x in the list y. *)

(NEXT-VS-OF-V V VERTICES NEXT-VS-SEQ)  
=  
(NTH (LOCATE V VERTICES) NEXT-VS-SEQ)

(* The function NEXT-VS-OF-VS  
   returns the union of all the  
   NEXT-VS's of the vertices in VS. *)

(NEXT-VS-OF-VS VS VERTICES NEXT-VS-SEQ)  
=  
(COND  
   ((LISTP VS)  
    (UNION  
      (NEXT-VS-OF-V (CAR VS) VERTICES NEXT-VS-SEQ)  
      (NEXT-VS-OF-VS (CDR VS) VERTICES NEXT-VS-SEQ))))  
   (T NIL))
Appendix E.

More Results from the Basic Experiment

Here are some more results of GT from the experiment described in chapter 4:

- The following three conditions are equivalent:

  (a) (ALL V VERTICES (NOT (MEMBER V (OUT-REACH V))))
  (b) (ALL V VERTICES (NOT (MEMBER V (IN-REACH V))))
  (c) (ALL V VERTICES (EQ (MIN-LOOP V)
                          (ADD1 NUM-VS))))

- The above conditions are equivalent when ALL is replaced by UNIQUE.

- The following two conditions are equivalent:

  (a) (ALL V VERTICES (ZEROP (IN-REACH-SIZE V)))
  (b) (ALL V VERTICES (ZEROP (OUT-REACH-SIZE V)))

- The above two conditions are equivalent when ALL is replaced by UNIQUE.

- (UNIQUE V VERTICES
  (AND (NOT (MEMBER V (OUT-REACH V)))
  (EQ (OUT-REACH-SIZE V) (SUB1 NUM-VS))))
  implies
  (UNIQUE V VERTICES (ZEROP (IN-DEGREE V))).
• The following conditions are equivalent:
  (a) (AND (ALL V VERTICES
       (NOT (MEMBER V (IN-REACH V))))
       (UNIQUE V VERTICES
       (AND (NOT (MEMBER V (OUT-REACH V)))
       (EQ (OUT-REACH-SIZE V) (SUB1 NUM-VS))))
  (b) (UNIQUE V VERTICES
       (EQ (IN-REACH-SIZE V) (SUB1 NUM-VS)))
  (c) (AND (ALL V VERTICES
       (NOT (MEMBER V (IN-REACH V))))
       (UNIQUE V VERTICES
       (ZEROP (IN-REACH-SIZE V))) t
Bibliography

[Boyer 79] Boyer, R. S. and Moore, J. S.
A Computational Logic.

[Buchanan 78] Buchanan B. G. and Feigenbaum E. A.
DENDRAL and Meta-DENDRAL.
Artificial Intelligence 11(1,2), 1978.

The Handbook of Artificial Intelligence (Volume 3).
Kaufman, Los Altos, California, 1982.

[Fu 85] Fu, L. M. and Buchanan, B. G.
Learning Intermediate Concepts in Constructing a
Hierarchical Knowledge Base.
In Proceedings of the 9th International Joint Conference on
Artificial Intelligence. Los Angeles, California, August, 1985.

[Kim 85] Kim, M. W.
EGS: A Transformational Approach to Automatic Example
Generation.
In Proceedings of the 9th International Conference on
Artificial Intelligence. Los Angeles, California, August, 1985.

[Langley 83] Langley, P. W.
Rediscovering Chemistry with the BACON System.
In Michalski, R. S., Carbonell, J. G., and Mitchell,
[Lenat 76] Lenat, D. B.
AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search.

[Lenat 83] Lenat, D. B.
Artificial Intelligence 21(1,2), 1983.

[Michalski 78] Michalski, R. S., and Larson, J. B.
Selection of most representative training examples and incremental generation of VL1 hypotheses: The underlying methodology and the description of programs ESEL and AQ11.

[Michalski 83a] Michalski, R. S. and Stepp, R. E.
Learning From Observation: Conceptual Clustering.

Machine Learning.
Tioga, Palo Alto, California, 1983.

[Michalski 83c] Michalski, R. S.
A Theory of Methodology of Inductive Learning.

[Minsky 75] Minsky, M.
A Framework for Representing Knowledge.
Mitchell, T. M.
Version spaces: A candidate elimination approach to rule
learning.
In Proceedings of the 5th International Joint Conference on
Artificial Intelligence. Los Angeles, California, August,
1977.

Popper, K. R.
Conjectures and Refutations: The Growth of Knowledge.

Porter, B. W. and Kibler, D. F.
Experimental Goal Regression: A Method for Learning
Problem Solving Heuristics.
1985.
To be published.

Rich, E.
Artificial Intelligence.

Robinson, D. F. and Foulds, L. R.
Digraphs: Theory and Techniques.

Simon, H. A. and Lea, G.
Problem Solving and Rule Induction: A Unified View.
In Gregg, L. (editor), Knowledge and Cognition. Lawrence

Simon, H. A.
Why Should Machines Learn?
In Michalski, R. S., Carbonell, J. G., and Mitchell,

Winston, P. H.
Learning structural descriptions from examples.
In Winston, P. H. (editor), The Psychology of Computer
VITA

Wing-Kwong Caesar Wong was born in Hong Kong on November 10, 1959, the son of May Kwan and Kwong Wong. After finishing his secondary and matriculation education in Wah Yan College, Hong Kong, in 1979, he entered the University of Toronto in Ontario, Canada. In 1981, he served as a teaching assistant at the Department of Computer Science. He received the degree of Bachelor of Science (specialist in computer science and major in mathematics) with distinction from the University of Toronto in 1982. In the fall of 1983, he entered the Department of Computer Science of Texas A&M University as a graduate student. In the spring of 1984, he transferred to the Department of Computer Science of the University of Texas at Austin. He served as a teaching assistant during the summer of 1984. Starting from the same semester, he has been serving as a research assistant and doing research in machine learning and expert systems.

Permanent Address: Flat 5, 29th Floor, Block E, King Shan Court, Ngau Chi Wan, Kowloon, Hong Kong.

This thesis was typed by the author.