Automatic Theory Formation in Graph Theory

HEMERSON PISTORI[1], JACQUES WAINER[2]

[1]Departamento de Engenharia de Computação, Universidade Católica Dom Bosco, Campo Grande, MS, Brasil
`pistori@lapac.unibosco.br`
[2]Instituto de Computação, Universidade Estadual de Campinas, Campinas, SP, Brasil
`wainer@dcc.unicamp.br`

Abstract. This paper presents SCOT, a system for automatic theory construction in the domain of Graph Theory. Following on the footsteps of the programs ARE [9], HR [1] and Cyrano [6], concept discovery is modeled as search in a concept space. We propose a classification for discovery heuristics, which takes into account the main processes related to theory construction: concept construction, example production, example analysis, conjecture construction, and conjecture analysis.

Keywords machine learning, theory refinement, constructive induction, unsupervised learning.

1 Introduction

Lenat's program AM was the first automatic theory construction system to attract the attention of AI community. Beginning with pre-numerical concepts, it "rediscovered" some well-known concepts and conjectures in elementary mathematics [7], such as the prime numbers, the deMorgan's Law and the Goldbach's conjecture.

AM's initial success and later inability to generate new results, were analyzed mainly by its creator. AM represented concepts by LISP programs that generated examples of the concept, and relied on the syntactic mutation of such code to generate new concepts. Part of the AM success was attributed to the close mapping between LISP code and mathematical concepts, so the mutation of the LISP code of a mathematical concept would very probably be a interesting mathematical concept. Another possible explanation for its success is that much of what it accomplished was in some way encoded into the 243 heuristics and 115 basic concepts [8].

The issue of concept representation in mathematical discovery systems was further pursued by Epstein [3]. She proposes a hierarchy of representational languages, the *R-languages*, for graph theory that uses declarative expressions. This representation scheme was used in the program GT (Graph Theorist), that was able to create new concepts, generate examples of this concepts, propose new conjectures, and prove theorems.

We implemented a program called SCOT, which is able to create new concepts, generate and analyze examples from concepts and propose conjectures. SCOT does not rely on mutations, like AM, to generate new concepts, but on the idea of operators being applied to one or two "old" concepts to generate a "new" concept. Furthermore, the language for the description of concepts is much more fine-grained than GT's. While GT worked only with Graph Properties and Graph Classes, SCOT treats uniformly both graph and other graph components. Thus SCOT can define concepts like cut vertex, which could not be discovered as such in GT because it is neither a Graph, nor a Graph Property (it's a vertex property).

Haase [6], Shen [9], and Bundy [1], propose a concept representation mechanism that is very similar to the one SCOT uses. It is, basically, a set of operators employed to create new concepts from old concepts, and a set of heuristics that determines the concepts and the operators to be used at each step. Given an initial set of concepts, and these operators, one can define concept discovery as the search of "interesting" concepts in this implicitly defined concept space. Now its possible to borrow some concepts from the search theory to concept discovery. Shen, for instance, applies an idea similar to that of macro operators to improve the concept search.

The operators in SCOT inherit some ideas from Backus' [2] work on functional programming, and are of a much finer grain than, for example HR's [1]. Furthermore, operators specialized for the graph theory domain were developed. In order to deal with the complexity of generating examples in the graphs domain, we designed SCOT to generate examples randomly, instead of systematically generating a large number of graphs, and to save some of these examples in a persistent database. When asked to generate another example, SCOT would randomly choose an example in the database or create a new one. Furthermore, the generation of examples of different concepts is run distributedly in many machines.

The issue of discovery heuristics seems to be dealt in more depth in [1]. That paper suggests some well-defined measures for concept and operators, such as clarity, parsimony, and novelty [1]. We used some of these measures and propose some new ones. Besides, we propose a classification for the discovery heuristics which relates to its representation mechanisms and scope of application, based on the following tasks: concept construction, example production, example analysis, conjecture construction and conjecture analysis.

The next section introduces the representation mechanism for the exploration domain. Then the main control structure and the heuristics are described. The last section presents some results, some problems and some suggestions for future work.

2 Domain Representation

The domain of exploration of our system is made up of concept, conjecture, example and operator objects. The set of concepts increases by the application of an unary or binary operator on a previous concept, or pair of concepts (which are called operands). Operators are closed related to Shen's *Functional Forms* and Bundy's *Production Rules*. Associated to each concept is a procedure that creates random examples of the concept, when required. The system guarantees that any concept, built-in or discovered, is always able to supply examples of itself. Conjectures are restricted to a simple relation between a pair of concepts. We describe now, in detail, each of SCOT's domain representation objects.

2.1 Concepts

The main attributes of a concept are *domain*, *type*, and *origin*. The domain of a concept X is another concept Y and it indicates that in order to obtain an example of X it is required an example of Y. For instance, the domain of the VERTEX DEGREE concept is VERTEX. While the domain of VERTEX is GRAPH. The GRAPH concept has no domain. The type of a concept determines the internal representation for examples of the concept. It is restricted to the following grammar (where S is the only non-terminal symbol):

$$\mathrm{S} \rightarrow \{\mathrm{S}\} \mid \{\mathrm{S}\}^n \mid [\mathrm{S}] \mid \mathrm{B} \mid \mathrm{I} \mid \mathrm{G} \mid \mathrm{V}$$

where B stands for Boolean, I for Integer, $\{\mathrm{S}\}$ for a set of type S elements, $\{\mathrm{S}\}^n$ for a set with fixed cardinality n, [S] stands for a sequence of type S elements, G stands for an undirected graph and V for a vertex.

Graphs are represented as a list where the first element is a sequence of integers representing the vertices and the second element is a sequence of pairs representing graph edges. For instance, the list $((123)((12)(13)(23)))$ represents the K_3 graph.

The origin of a concept indicates if the concept is a built-in concept or if it was discovered by the system. In a built-in concept, the origin carries a predefined code, which is called when an example of the concept is requested. In a discovered concept, the origin holds the operator and operands (concepts), used in its creation. This information is used in the example generation process.

Built-in concepts are divided into graph specific concepts and logical/numeric concepts. The graph specific concepts are: a graph, a vertex (of a graph), all vertices (of a graph), a pair of vertices (of a graph), all pair of vertices (of a graph), an edge (of a graph), all edges (of a graph), neighborhood (of a vertex), all paths from a vertex, and a vertex-set induced subgraph (of a graph). The majority of the logical/numeric concepts are tests on boolean or integer sequences, like: all boolean values are true, there is no true value in the sequence, all integers are greater than zero, and so on. SCOT never combines two "non-graph" concept, so that it can never "escape" from the graph domain.

2.2 Examples

An example is an object with a *content* attribute, a *domain example* attribute (which holds an example of the concept domain, when it is the case) and a *concept* attribute (linking examples to concepts). Figure 1 shows examples of 4 related concepts (from right to left): GRAPH, ALL EDGES, ALL CUT EDGES CLASSIFIER[1] and ALL CUT EDGES. The GRAPH concept generates a random graph example. The ALL EDGES takes this graph example as its domain example and generates an example containing all the edges of the graph. The ALL CUT EDGES CLASSIFIER concept does not have a direct counterpart in graph theory. It just maps each edge of a graph to a boolean that indicates if the edge is a cut edge (in that graph). The origin of this concept indicates that it is the application of the ATA (Apply To All) operator on the concept 2 (Not shown in figure 1) and 3 (ALL EDGES). Concept 2 is the CUT EDGE CLASSIFIER, which is of type boolean (as any classifier) and has the concept AN EDGE (of a graph) as its domain. The ATA operator is used to apply a particular boolean type concept (Eg. CUT EDGE CLASSIFIER) to each element of a set (or sequence) (Eg. The content of a ALL EDGES example).

The ALL CUT EDGES selects from all the graph's edges the ones that are mapped to "true" by the ALL CUT EDGES CLASSIFIER example. The origin of the ALL CUT EDGES concept indicates that it is the application of the INVT operator on the ALL CUT EDGES CLASSIFIER concept. The INVT exploit the "domain chain" in order to build examples. For instance, to construct an example for the ALL CUT EDGES concept, the INVT operator which is the origin of the concept, needs not only its domain example (the ALL CUT EDGES CLASSIFIER example – a list of booleans) but also its domain's domain example (a list of pairs of vertices – the edges). INVT selects from the domain's domain example all the items that are mapped to a "true" in the domain example.

[1] A classifi er is a boolean type concept. It resembles what Epstein has called a tester [4]

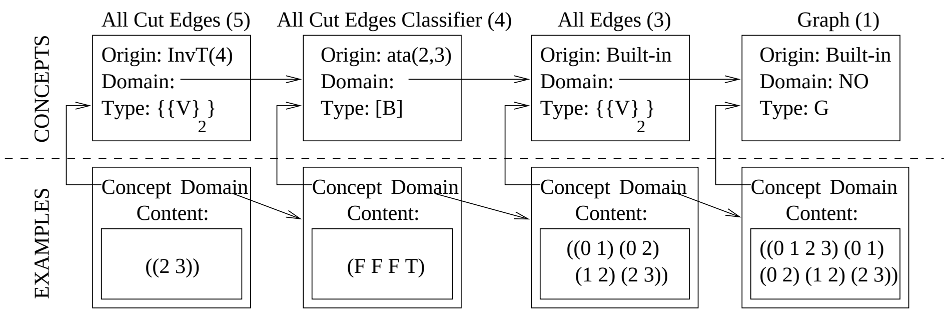

Figure 1: Examples and Concepts

2.3 Operators

The basis for SCOT capabilities to create new concepts is the operators, which are extensions of the functional forms introduced by Backus [2]. Each operator must implement three functionalities: (1) The *Applicability Verification* that verifies if the operator can be applied to a concept (or pair of concepts, in the case of binary operators), (2) The *Concept Generation* that applies the operator to create a new concept and (3) the *Example Generation* that generates examples for the new concepts.

We have created a diagram to represent concepts, which summarizes all the information in figure 1. Figure 2 is the diagrammatic representation of those concepts, where circles represent the concepts (dashed-lines denoting built-in concepts), the labels in the circles represent the type of the concept, a bold line indicate the domain relation, and a thin line represents the operator application. For example, concept 4 (ALL CUT EDGES CLASSIFIER) in figure 2 has as input (domain) the concept 3, its type is a set of booleans, and it is created by the application of the operator ATA on concepts 2 and 3. The two applications of the composition operator is used to adjust the domain of the ALL CUT EDGES concept to be the GRAPH concept. The diagram is not complete since it does not show neither the domain nor the origin of concept 2, which would have required the addition of several new circles and lines (Complete information about concept 2 could be represented hierarchically, in another diagram).

2.4 Macro-Operators

We have observed that some sequences of operators are common in the generation of interesting concepts. We call those templates of application of concepts a macro-operator. The same notation introduced in the last section (Figure 2) can be used in the visualization of a macro-operator. For instance, one of the macro-operators used by SCOT is the one described in figure 2 when concepts 2 and 3 are taken as variables. This macro-operator takes two concepts such that they can be the operands of the ATA operator and after a sequence of operators (ATA, INVT and 2 Compositions) application creates a concept of the same type and domain as the second argument. Macro-operators empower SCOT with the capability to construct complex concepts (many operators) in one single step.

2.5 Conjectures

A conjecture object holds a relation between two classifiers. These classifiers, that must have the same domain, are called the *Left* and the *Right* concepts. SCOT generates a set of examples of the domain of these two classifiers, and applies both of these concepts to classify each example of this set. Based on this results, SCOT *conjecture induction heuristic* (Explained in the Conjecture Analysis section) infers one of the following relations:

Equality The $Left$ concept returns *t*rue if and only if, the $Right$ concept returns *t*rue

Left implies Right The $Right$ concept always returns *t*rue when the $Left$ returns *t*rue. Symmetrically, $Right$ implies $Left$.

Undetermined No previous relation applies but there are some domain examples for which both the $Left$ and the $Right$ return *t*rue.

No relation No previous relation applies.

3 Control and Heuristics

SCOT runs cyclically its five modules: *concept construction, example analysis, conjecture construction, conjecture analysis* and *conjecture refinement*, until a time limit is reached. We call the execution of the five modules a *cycle*, and the execution of as many cycles as possible within the maximum time limit a *run*. All concepts and examples created in a cycle are available in all other cycles. At the end of the run, the “interesting” concepts and conjectures discovered and their examples are saved to a file. A new run may read in data saved by old runs and thus start with more than the initial concepts.

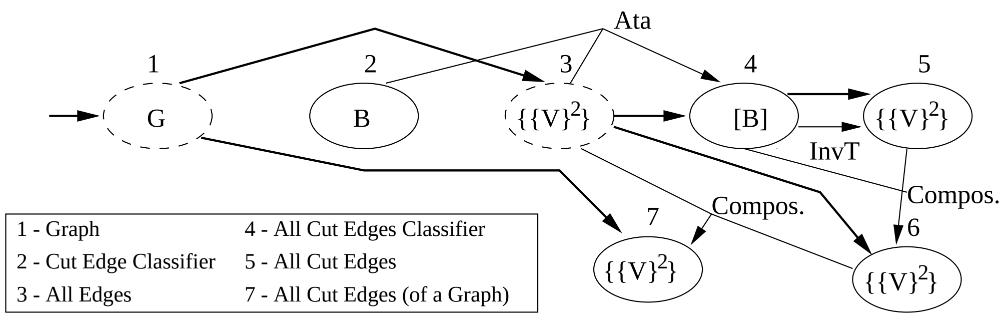

Figure 2: All cut edges concept

The example and conjecture analysis modules may be run distributedly, each machine dealing with a disjoint set of concepts or conjectures. In this way, all example generation overhead, which is usually a very costly process in graph theory domain, may be lessened.

We have seven distinct groups of heuristics, each group with a specific representation mechanism: *macro-operators*, *operator evaluation function*, *concept evaluation function*, *example evaluation function*, *conjecture induction heuristic*, *conjecture refinement heuristic* and the *numeric thresholds*. The first 6 groups where explained previously in this paper and the numeric thresholds are numbers like the maximum number of concepts that can be generated in a cycle, the maximum number of examples to be generated in the example analysis module, and so on.

There are still some "heuristic reasoning" that are not in these groups. They are related to how one selects the concepts and operators that are to appear in the initial knowledge database. For instance, numbers greater than 3 never happen in external concepts (resembling an heuristic used by Lenat), in this way, we have in our initial database the concepts ALL INTEGERS EQUALS 0, ALL INTEGERS EQUALS 1, ..., ALL INTEGERS EQUAL 3, but not the concept ALL INTEGERS EQUALS 4.

3.1 Concept Construction

The concept construction module applies operators and macro-operators to create new concepts, until a time limit is reached. Each operator and macro-operator has a numerical value that determines the probability that they will be chosen. Concepts on which the operators will be applied are chosen based on a numeric value called *concept worth*, which is dynamically calculated in each cycle by the *concept evaluation function*. This function combines some other numeric values associated with the concept, such as: the number of conjectures involving the concept, the mean time for generating examples for this concept, and the complexity of the concept's generation tree (that is the number of operators and other concepts that make up this concept). The two other values taken into consideration by this evaluation function are a *user worth* value, which allow the experimenter to focus the system into a set of concepts, and the *examples worth* which is calculated by the example analysis module and reflects how "interesting" are the examples for this concept. The *concept worth* value is also used in the Conjecture Construction module and in the selection of the concepts to be saved in the end of a run.

3.2 Example Analysis

The example analysis module generates the examples for the concepts created in the previous module, and analyse these examples. Example generation is an expensive operation in SCOT because it uses a generate and test approach: all examples end up referring to a graph in its domain chain, so a random graph is generated and all further operations will either manipulate that graph or refuse it (because it does not satisfy a property), and then a new random graph must be generated and tested. There is an heuristic number of how many times will the module try to generate an example of a concept before giving up.

When 40 (another *numeric threshold*) examples of the concept have been generated, the module will determine how "interesting" are these examples and set the concept's example worth value. There are some simple heuristics to determine if a single example and if a set of examples are interesting. For example, if all examples of a concept have the same content, that is, the set of examples really contain only one example, the concept will receive a very low example-worth value. If the system could not generate a single example for that concept in the allowed time, the concept will receive the lowest example worth value. If all examples of a graph concept are empty graphs (graphs without edges) then the concept will also receive a very low example-worth value.

3.3 Conjecture Construction

Conjecture construction is a rather simple module that chooses pairs of classifiers with the same domain as candidates for a conjecture. Concepts with higher *concept*

Figure 3: Conjecture Induction Heuristic

Relation	TT	TF	FT
Equivalence	$> 15\%$	$= 0\%$	$= 0\%$
R implies L	$> 2\%$	$> 10\%$	$= 0\%$
L implies R	$> 2\%$	$= 0\%$	$> 10\%$
Undetermined	$> 2\%$	$> 0\%$	$> 0\%$
No Relation	None of	the previous	applies

worth are favored in this selection process. After a fixed number of such conjectures have been constructed, the conjecture analysis phase is started.

3.4 Conjecture Analysis

The conjecture analysis module tries to infer, for each conjecture, the relation between its two classifiers. It generates 100 examples of the common domain concept and calculates the relative number of examples from the set that is classified as "true" by both classifiers (TT), and the number of examples that are classified as "true" by one but not by the other classifier (TF and FT).

Given the TT, TF and FT values, the *Conjecture Induction Heuristic* (represented in figure 4) proposes a relation for the two classifiers in the concept. For example if more than 2% of the examples were classified as "true" by both concepts, and more than 10% of the examples were classified as "true" by the $Left$ concept but as false by the $Right$ concept, and none was classified as "true" by $Right$ and "false" by $Left$, then the system will conjecture that all examples of the $Right$ concept are also examples of the $Left$ concept (R implies L).

3.5 Conjecture Refinement

The conjecture refinement module creates new conjecture objects substituting a concept of the pair for some of its specializations or generalizations. The conjecture refinement goal is to increase the number of equivalence and implication conjectures. This module may also request the creation of generalizations or specializations of some concepts. This requests are kept in an agenda, accessible to all modules. Thus if a specialization of a concept is placed into the agenda, the concept creation module will first attempt to comply with the request before creating "random" concepts. Similarly the conjecture construction module will first try to construct the conjectures that involves the specialization that was requested.

Besides the implications discovered empirically in the conjecture analysis module, SCOT has another mechanism, based on macro-operators, to find specializations and generalizations. Relations imposed by macro-ope-ra-tors are logical and not empirical. For instance, the concept created by the macro-operator described in figure 2 (with concepts 2 and 3 taken as variables) is always a specialization of the concept that is placed on the position of concept 3 (In that example, ALL CUT EDGES is a specialization of ALL EDGES). Thus the first form of specializing a concept is applying such a macro-operator.

The *conjecture refinement* heuristic is a function that, given two conjecture objects, decides which is the most "valuable". It evaluates the relation and the TT,TF and FT values of each conjecture. If the relation of the two conjecture is not the same, the heuristic uses the following ordering to decide which is the better one: equality $>$ implication $>$ undetermined $>$ no relation. If the relation is the same, the system decides for the conjecture with the lesser $TF + FT$ value, which roughly indicates that the conjecture is nearer the equality relation. The conjecture refinement module uses this heuristic in order to evaluate if a given substitution should be done.

3.6 An example

To illustrate the operation of SCOT, we will show how the modules and the specialization/generalization mechanism were used to propose the conjecture that all cycle graphs are regular.

- Applying a macro-operator M_1 on ALL PATHS (leaving a vertex) and PAIR OF VERTICES the system discovers the concept ALL PATHS CONNECTING TWO VERTICES.
- Applying a macro-operator M_2 on NEIGHBORHOOD (of a vertex) SCOT gets the VERTEX DEGREE concept.
- Applying a macro-operator M_3 on ALL PATHS CONNECTING TWO VERTICES, ALL PAIR OF VERTICES and ALL INTEGERS GREATER THAN 0 the system gets the CONNECTED GRAPH (Graphs where there is at least one path connecting each pair of vertices) classifier.
- Substituting the concept ALL INTEGERS GREATER THAN 0 for its specialization, ALL INTEGERS EQUAL TO 1, in the definition of CONNECTED GRAPH, the system generates the CYCLE GRAPH (Graphs where there is one and just one path connecting each pair of vertices [2]) classifier, as a specialization of the former.
- Applying the same M_3 on VERTEX DEGREE, ALL VERTICES (of a graph) and ALL INTEGERS EQUAL TO EACH OTHER, the system creates the REGULAR GRAPH classifier.

[2]This defi niton, although unusual, resembles more closely that of SCOT

- The system creates a conjecture object C_1 involving the connected and regular classifier and concludes for an undetermined relation ($TT = 3\%$, $TF = 35\%$, $FT = 11\%$).
- Trying to refine C_1, the system creates a new conjecture by substituting CONNECTED GRAPH by the CYCLE GRAPH (known, by SCOT, as an specialization of the former). Analyzing this new conjecture SCOT concludes for a "left implies right" relation ($TT = 7\%, TF = 0\%$, $FT = 16\%$).

These steps were not taken one immediately after the other, but were taken in that order intermingled with other steps, in a single run.

4 Conclusions

The programs ARE [9] and Cyrano [6] work in the same domain as AM (number theory and arithmetic), HR [1] addresses Group Theory and Number Theory. Although, theoretically all these programs could be adapted to work in other domains, SCOT seems to be the first system, of this kind, whose natural domain of exploration is Graph Theory (Grafitti [5] and GT [3] work in Graph Theory but taking a very different approach). SCOT also seems to be the first system to use distributed computation in order to improve its performance.

Another improvement is related to the way heuristics are organized and represented in SCOT. We have identified 6 very distinct classes of heuristics, each with rather particular characteristics. For instance, the *macro-ope-ra-tors* deals only with structural properties of concepts, whereas the *example analysis heuristics* (summarized in a single number by the example evaluation function) works with properties of concept examples. The identification of these classes allowed for the utilization of a more restrictive (and structured) representation mechanism (for each class), without much loss in expressiveness.

In a 8-hour run, using 14 machines for the distributed modules, SCOT would generate around 600 to 800 concepts with non-zero worth value (SCOT begins with a set of 15 graph theory concepts, 30 logic and numerical concepts, 17 operators and 6 macro-operators). We recognized among the concepts generated by SCOT many human relevant concepts such as complete, regular, and empty graphs, cycle graphs, connected graphs, cut edges, trees, and so on (in fact, we coded those concepts in SCOT's language and searched for them in the files of concepts generated). We also recognized conjectures involving tho-se concepts, such as "all trees are connected" and "all complete graphs are regular."

An interesting line for future research would be the assumption that SCOT and maybe other discovery systems should really work in a supervised way, not as an autonomous discoverer, but as a helper for researchers in a particular domain. We are currently tuning SCOT so that it centers its explorations around the concepts that a researcher in graph theory is interested in. In this supervised mode, SCOT would generate few concepts and examples for those concepts, and wait for an external evaluation of their "interestingness" before combining them into yet other concepts.

A second line of research is based on the adaptation of some pattern recognition techniques for the automatic discovering of new macro-operators. First of all, we must build a large database of well-known graph theory concepts, represented using operators and built-in concepts. Then, we must search these database for some pattern (macro-operator) in the operators application (These problem is worsened by the fact that different groups of concepts can induce different patterns). Luckily, these newly discovered macro-operators could be used by SCOT to produce interesting concepts that do not appear in the large database.

References

[1] S. Colton A. Bundy and T. Walsh. HR - a system for machine discovery in finite algebras. *ECAI 98 Workshop Programme*, 1998.

[2] John F. Backus. *ACM Turing Award Lectures, the first twenty years*, chapter Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs, pages 63–130. Addison-Wesley, 1987.

[3] S. L. Epstein. Learning and discovery: One system's search for mathematical knowledge. *Computational Intelligence*, 4-1, 1988.

[4] S. L. Epstein and N. S. Sridharan. Knowledge representation for mathematical discovery: Three experiments in graph theory. *Applied Intelligence*, 1, 1991.

[5] S Fajtlowicz. On conjectures of graffiti. *Discrete Mathematics*, 23:113–118, 1998.

[6] K. Haase. Discovery systems. *Proceedings of the 7th European Conference on Artificial Intelligence*, 1986.

[7] Douglas B. Lenat. Why AM and EURISKO appear to work. *Journal of Artificial Intelligence*, 23, 1984.

[8] G. D. Ritchie and F. K. Hanna. AM: A case study in AI methodology. *Journal of Artificial Intelligence*, 23, 1984.

[9] W. Shen. Functional transformations in AI discovery systems. *Artificial Intelligence*, 41, 1989.