# Concept Formation Using Graph Grammars

**Istvan Jonyer, Lawrence B. Holder and Diane J. Cook**

Department of Computer Science and Engineering
University of Texas at Arlington
Box 19015 (416 Yates St.), Arlington, TX 76019-0015
E-mail: {jonyer | holder | cook}@cse.uta.edu
Phone: (817) 272-2596, Fax: (817) 272-3784

## Abstract

Recognizing the expressive power of graph representation and the ability of certain graph grammars to generalize, we attempt to use graph grammar learning for concept formation. In this paper we describe our initial progress toward that goal, and focus on how certain graph grammars can be learned from examples. We also establish grounds for using graph grammars in machine learning tasks. Several examples are presented to highlight the validity of the approach.

## Introduction

Graphs are important data structures because of their ability to represent any kind of data. Algorithms that generate theories of graphs are of great importance in data mining and machine learning. In this paper we describe an algorithm which learns graph grammars—a set of grammar production rules that describe a graph-based database.

The goal of our research is to adapt graph grammar learning for concept formation, hoping that the expressive power of graphs and the ability of graph grammars to generalize will turn out to be a powerful learning paradigm. This paper presents initial progress toward that goal and sets the stage for subsequent work.

Only a few algorithms exist for inference of graph grammars. An enumerative method for inferring a limited class of context-sensitive graph grammars is due to Bartsch-Spörl (1983). Other algorithms utilize a merging technique for hyperedge replacement grammars (Jeltsch and

Kreowski 1991) and regular tree grammars (Carrasco et al. 1998). Our approach is based on a method for discovering frequent substructures in graphs (Cook and Holder 2000).

In the following section we discuss different types of graph grammars and argue how they can be useful in machine learning. We then describe the graph grammars we set out to learn and define some terminology. Next, we present a set of examples to provide some visual insight with graph grammars before we describe the algorithm. Then, we present a working example on an artificial domain to better illustrate the algorithm. Next, we discuss the types of grammars the algorithm can learn, as well as point out some of its limitations. We conclude with an overall assessment of the approach and give directions for future work.

## Graph Grammars and Machine Learning

When learning a grammar in general, one has to decide the intended use of the grammar. Grammars have two applications: to parse or to generate a language. Parser grammars are optimized for fast parsing, giving up a little accuracy which results in grammars that over-accept. That is, they will accept "sentences" that are not in the language. Generator grammars trade accuracy for speed as well. As expected, they will not be able to generate the entire language. A grammar that can generate and parse the same language exactly is very hard to design and is usually too big and slow to be practical.

In this paper we are addressing the problem of inferring graph grammars from positive examples. Our purpose is to use grammar learning as an approach to data mining, but other uses can also be found. The generated graph grammar will be our theory of the input domain.

In machine learning, algorithms in general are attempting to learn theories that can generalize to a certain degree, so that new, unseen data can be accurately categorized. Translated to grammar terms, we would like to learn a grammar that accepts more than just the training language. Therefore, we would like to learn parser grammars, which have the power to express more general concepts than the sum of the positive examples.

Grammars can be context-sensitive and context-free. Context-sensitive graph grammars are more expressive and allow the specification of graph transformations, since both sides of the production can be arbitrary graphs. To start with, however, we aimed at learning context-free grammars that have single-vertex non-terminals on the left side of production rules. This is not a serious limitation, especially since the vast majority of graph grammar parsers can only deal with exactly such grammars (Rekers and Schürr 1995).

So why learn graph grammars versus textual ones? Textual grammars are also useful, but they are limited to databases that can be represented as a sequence. An example of such a database is a DNA sequence. Most databases, however, have a non-sequential structure, and many have significant structural components. Relational databases are generally good examples, but even more complex information can be represented using graphs. Examples include circuit diagrams and the world-wide web. Graph grammars can still represent the simpler feature vector-type databases as well as sequential databases (like the DNA mentioned previously). Graphs are among the most expressive representations, therefore an algorithm that can learn a theory of a graph would be useful.

We have to emphasize that our purpose in learning graph grammars is not to provide an efficient graph parsing algorithm. Graph parsing will be necessary for classifying unseen example graphs, and while the parsing efficiency of the graph grammar will be a concern here, it is not a primary goal of the generalization step.

## Graph Grammars

Before we get into the details of inferring graph grammars, we first give a general overview of the type of grammar we seek to learn.

In this paper, and in our research in general, we are concerned with graph grammars of the set theoretic approach, or expression approach (Nagl 1987). In this approach a *graph* is a pair of sets $G = \langle V, E \rangle$ where $V$ is the set of *vertices* or *nodes*, and $E \subseteq V \times V$ is the set of *edges*. Production rules are of the form S → P, where S and P are graphs. When such a rule is applied to a graph, an isomorphic copy of S is removed from the graph along with all its incident edges, and is replaced with a copy of P, together with edges connecting it to the graph. The new edges are given new labels to reflect their connection to the substructure instance.

A special case of the set-theoretic approach is the node-label controlled grammar, in which S consists of a single labeled node (Engelfriet and Rozenberg 1991). This is the type of grammar we are focusing on. In our case, S is always a non-terminal, but P can be any graph, and can contain both terminals and non-terminals. Since we are going to learn grammars to be used for parsing, the embedding function is irrelevant. External edges that are incident on a vertex in the subgraph being replaced (P) always get reconnected to the single vertex S.

Recursive productions are of the form S → P S. The non-terminal S is on both sides of the production, and P is linked to S via a single edge. The complexity of the algorithm is exponential in the number of edges considered between recursive instances, so we limit the algorithm to one for now. If the grammar is used for graph generation, this rule will generate an infinitely long sequence of the graph P. If the language is to be finite, a stopping alternative production is required. One such production is S → P S | ∅, which reads "replace S with P S or nothing." For our purposes, however, we use the production S → P S | P. The rule S → P S | ∅, when used for parsing, would imply that nothing can be replaced with S, introducing an arbitrary number of S's. At the same time, it cannot parse a chain of P's of finite length as it would have no starting point, since P S does not exist in the input graph. Remember that the stopping alternative of a graph generator rule is the starting point of a parser rule.

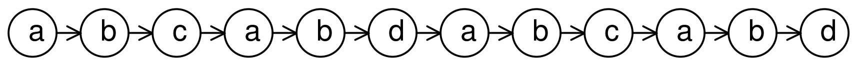
When parsing a graph, we start from the complete graph and work towards a single non-terminal. This is done by removing subgraphs from the graph that match the right side of a production and inserting the non-terminal on the left side—in our example, replace P S with S, and finally, P with S. An example of a recursive production is shown in Figure 1c, ($S_1$).

Alternative productions are of the form $S \rightarrow P_1 \mid P_2 \mid \ldots \mid P_n$. The non-terminal graph S can be thought of as a variable having possible values $P_1$, $P_2$, …, $P_n$. We will sometimes refer to such an S as a variable non-terminal, or simply variable. If S is a single vertex, and $P_i$ are also single vertices, then S is synonymous with a regular non-graph variable. Its values are the vertex labels, which can be alphanumeric values like numbers (discrete or continuous) or string descriptions. An example of a variable is shown in Figure 1c, where $S_2$ has possible values 'c' and 'd'.

## Examples

Before presenting the algorithm, a couple of examples are given here to further clarify what we are trying to accomplish. The first example is suggested by the authors of Sequitur (Nevill-Manning and Witten 1997). Sequitur infers compositional hierarchies from strings. It detects repetition and factors it out by forming rules in a grammar.

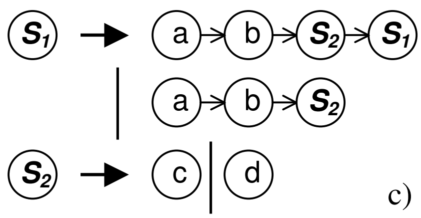
The example string to be analyzed is "abcabdabcabd". The grammar generated by Sequitur is shown in Figure 1a. (Non-terminals are in italic bold font.) Our algorithm, called SubdueGL, learns graph grammars; therefore, the input has to be in a graph format. This sequential data was represented by a series of vertices having labels according to the example, connected by single edges, as shown in Figure 1b. The graph grammar learned by SubdueGL is shown in Figure 1c, while its sequential interpretation is shown in Figure 1d. The first obvious difference is that SubdueGL is able to learn recursive grammars. SubdueGL's version of the grammar is also more general, since it would parse a string of any length, and the letters 'c' and 'd' do not have to follow in the same order. This example will be referenced in the next section where we describe the algorithm.




b)

$\mathbf{S} \rightarrow \mathbf{1}\ \mathbf{1}$
$\mathbf{1} \rightarrow \mathbf{2}\ \mathrm{c}\ \mathbf{2}\ \mathrm{d}$
$\mathbf{2} \rightarrow \mathrm{a}\ \mathrm{b}$

a)




c)

$\mathbf{S_1} \rightarrow \mathrm{a}\ \mathrm{b}\ \mathbf{S_2}\, \mathbf{S_1}$
$\quad |\ \mathrm{a}\ \mathrm{b}\ \mathbf{S_2}$
$\mathbf{S_2} \rightarrow \mathrm{c}\ |\ \mathrm{d}$

d)

**Figure 1** First example: a) Grammar by Sequitur b) Input graph to SubdueGL c) Graph grammar by SubdueGL d) Equivalent string grammar

The next example is a variation of the previous one, with an 'x' slightly breaking the regularity in the pattern: "abcabdxabcabd". The grammar learned by Sequitur is shown in Figure 2a and is very similar to the previous one in Figure 1a. SubdueGL, however, added an extra production to its grammar, resulting in the grammar shown in Figure 2b.

$\mathbf{S} \rightarrow \mathbf{1}\ \mathrm{x}\ \mathbf{1}$
$\mathbf{1} \rightarrow \mathbf{2}\ \mathrm{c}\ \mathbf{2}\ \mathrm{d}$
$\mathbf{2} \rightarrow \mathrm{a}\ \mathrm{b}$

a)

$\mathbf{S_1} \rightarrow \mathrm{a}\ \mathrm{b}\ \mathbf{S_2}\, \mathbf{S_1}$
$\quad |\ \mathrm{a}\ \mathrm{b}\ \mathbf{S_2}$
$\mathbf{S_2} \rightarrow \mathrm{c}\ |\ \mathrm{d}$
$\mathbf{S_3} \rightarrow \mathbf{S_1}\ \mathrm{x}\ \mathbf{S_1}$

b)

**Figure 2** Grammars by a) Sequitur and b) SubdueGL learned from "abcabdxabcabd".

A third example is given later, after the introduction of the algorithm. That example involves an artificial domain which was specifically designed to highlight SubdueGL's capabilities.

## The Learning Algorithm

The SubdueGL algorithm is based on the Subdue (Cook and Holder 2000) knowledge discovery algorithm that extracts common substructures from graphs. SubdueGL takes data sets in a graph format as input, hence a database needs to be represented as a graph before passing it to SubdueGL. The graph representation includes the standard features of graphs: labeled vertices and labeled edges. Edges can be directed or undirected. No restrictions are placed on the input graph. When converting a data set to a graph representation, typically objects and data are mapped to vertices, and relationships and attributes are mapped to edges.

### *Search*

SubdueGL performs an iterative search on the input graph, each iteration resulting in a grammar production. When a production is found, the right side of the production is abstracted away from the input graph by replacing each occurrence of it by the non-terminal on the left side. SubdueGL keeps iterating until the entire input graph is abstracted away into a single non-terminal. User-specified limits can be placed on the number of productions to be found, and on the maximum size of each production rule (in number of vertices). For other user-defined options please see earlier publications, or the user manual, available at http://cygnus.uta.edu/subdue/.

In each iteration SubdueGL performs a beam search for the best substructure to be used in the next production rule. The search starts by finding each uniquely labeled vertex and all their instances in the input graph. In our first example the input graph (shown in Figure 1b) has 4 uniquely labeled vertices 'a', 'b', 'c' and 'd', each having 4, 4, 2 and 2 instances, respectively. The subgraph definition and all instances are referred to as a substructure. The

*ExtendSubgstructure* search operator is applied to each of these single-vertex substructures to produce 2-vertex substructures. This operator extends a substructure in all possible directions, and collects instances that match, possibly resulting in several new substructures. In our example, extending the unique vertex will result in instances like 'ab', 'ca' and 'da'. These are different from each other, but each have several instances of their own. These instances are collected to form new substructures.

```
SubdueGL ( graph G, int Beam, int Limit )
  repeat
    grammar = {}
    queue Q = { v | v has a unique label in G }
    bestSub = first substructure in Q
    repeat
      newQ = {}
      for each substructure S in Q
        newSubs = ExtendSubstructure(S)
        recursiveSubs = RecursifySubstructure(S)
        newQ = newQ U newSubs U recursiveSubs
        Limit = Limit - 1
      evaluate substructures in newQ by MDL
      Q = substructures in newQ with top Beam compression
           scores
      if best substructure in Q better than bestSub
      then bestSub = best substructure in Q
    until Q is empty or Limit <= 0
    grammar = grammar U bestSub
    G = G compressed by bestSub
  until bestSub cannot compress the graph G
  return grammar

ExtendSubstructure (substructure S)
 newSubs = S extended by an adjacent edge in all possible ways
 varSubs = S extended by an adjacent edge in all possible ways,
             replacing the added vertex with a non-terminal
 return newSubs U varSubs

RecursifySubstructure (substructure S)
 recSubs = all possible chains of instances of S, linked by a
             single edge
 return recSubs
```

**Figure 3** SubdueGL's main discovery algorithm.

The resulting substructures are evaluated according to the minimum description length (MDL) principle, which states that the best theory is the one that minimizes the description length of the entire data set. The MDL principle was introduced by Rissanen (1989), and applied to graph-based knowledge discovery by Cook and Holder (1994). The "value" of a substructure is computed by dividing the description length of the input graph by the sum of the description lengths of the substructure and the input graph compressed by the substructure: Value(S) = DL(G) / (DL(S) + DL(G|S)), where $G$ is the input graph, $S$ is the substructure, and $G \mid S$ is the input graph $G$ compressed using $S$. DL stands for "description length." SubdueGL seeks to maximize the value. Only substructures deemed the best by the MDL principle are kept for further extension.

Heuristics are applied to stop the extension process, although exhaustive analysis is also possible. One heuristic involves tracking the search space for local minima. The search is abandoned if a new local minimum is not found after a certain number of applications of the *Extend-Substructure* operator. This heuristic is used by default. Another heuristic requires the user to specify the maximum size of a substructure. Once the size limit is reached, the search terminates. There also exists an absolute limit on the number of substructures to consider during the search process, which can be specified by the user. These heuristics, like others in SubdueGL, can also be used in combination.

***Recursion***

Recursive productions are made possible by the *Recursify-Substructure* search operator. It is applied to each substructure after the *ExtendSubstructure* operator. *Recursify-Substructure* checks each instance of the substructure to see if it is connected to any other instance of the same substructure by an edge. If so, a recursive production is possible. The operator adds the connecting edge to the substructure and collects all possible chains of instances. If a recursive production is found to be the best in an iteration, each such chain of subgraphs is abstracted away by replacing them with a single vertex.

Since SubdueGL discovers commonly occurring substructures first and then attempts to make a recursive production, SubdueGL can only discover recursive productions that parse lists of substructures. In other words, it can only make recursive productions out of lists of substructures that are connected by a single edge, which have to have the same label between each member substructure of the list. The algorithm is exponential in the number of edges considered in the recursion, so we limit SubdueGL to single-edge recursive productions. Therefore, the system does not yet learn productions such as S → aSb.

The stopping condition in the recursion is generated by removing the recursive vertex along with the edge that connects it to the rest of the subgraph.

***Variables***

The major insight behind discovering variables is that the context they appear in has to be constant. In other words, the first step towards discovering variables is discovering commonly occurring structures. If these commonly occurring structures are connected to varying vertices, these varying vertices can be turned into variables. Variables have to be connected to each instance of the common substructure the same way—connected by an edge of the same label and direction to the same vertex. We give an example of this in the next section, but looking at the input graph in Figure 4 and the grammar rule in Figure 7 at this point could be helpful.

SubdueGL discovers variables inside the *ExtendSub-structure* search operator. As mentioned before, SubdueGL extends each instance of a substructure in all possible ways and collects the instances that still match. After this step, it collects all instances that were extended by the same edge, regardless of what vertex they point to (as long as that vertex is not already in the substructure). This new vertex, is replaced with a variable (non-terminal) vertex. The substructure is then evaluated and competes with others for top placement. Generally speaking, if the variable has the same value for most of the instances, the substructure will rank worse than the equivalent structure without the variable because of the extra bits needed to encode the variable. On the other had, if the variable has many values, it helps to cover many more instances of the substructure than the equivalent structure without the variable, and will compress the input graph better.

Let us work an example to illustrate the above explanation.

## An Illustrative Example

In this section we give a working example of SubdueGL's operation. Consider the input graph shown in Figure 4. It is the graph representation of an artificially generated domain. It features lists of static structures (square shape), a list of a changing structure (triangle shape), and some additional random vertices and edges. For a cleaner appearance we omitted edge labels in the figures. The edge labels within the triangle-looking subgraph are 't', in the square-looking subgraph 's', and the rest of the edges are labeled 'next'.

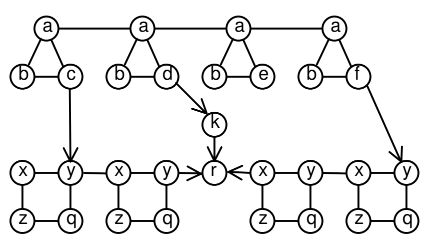



**Figure 4** Input graph.

SubdueGL starts out by collecting all the unique vertices in the graph and expanding them in all possible directions. Let us follow the extension of vertex 'x'—keeping in mind that the others are expanded in parallel. When we expand vertex 'x' in all possible directions, it results in 2-vertex substructures, with edges (x, s, y), (x, s, z), (y, next, x), and (x, next, r). The fist two substructures will rank higher, since those have four instances and will compress the graph better than the latter two with only 2 and 1 instances respectively.

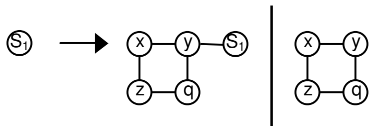



**Figure 5** First production generated by SubdueGL

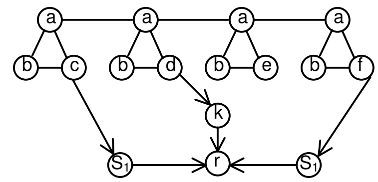



**Figure 6** Input graph, parsed by the first production

Applying the *ExtendSubstructure* operator three more times will result in a substructure having vertices {x, y, z, q} and four edges connecting these four vertices. This substructure has four instances. Being the biggest and most common substructure, it will rank on the top. Executing the *RecursifySubstructure* operator will result in the recursive grammar rule shown in Figure 5. The production covers two lists of two instances of the substructure.

The recursive production was constructed by checking all outgoing edges of each instance to see if they are connected to any other instance. We can see in Figure 4 that the instance in the lower left is connected to the instance on its right, via vertex ‘y’ being connected to vertex ‘x’. Same is the situation on the lower right side. Abstracting out these four instances of the substructure using the above production results in the graph depicted in Figure 6.

The next iteration of SubdueGL will use this graph as its input graph to learn the next grammar rule. Looking at the graph, one can easily see that the most common substructure now is the triangle-looking subgraph. SubdueGL will in fact find a portion of that simply by looking for substructures that are exactly the same. This part is the substructure having vertices {a, b} and edge (a, t, b). It has four instances. Extending this structure further by an edge and a vertex will add different vertices for each instance: ‘c’, ‘d’, ‘e’, and ‘f’. The resulting single instance substructures will not do well when evaluated by the MDL measure.

SubdueGL at this point will generate another substructure with four instances, replacing vertices ‘c’, ‘d’, ‘e’, and ‘f’ with a non-terminal vertex ($S_3$) in the substructure, thereby creating a variable. This substructure now has four instances, and stands the best chance of getting selected for the next production.

After the *ExtendSubstructure* operation, however, SubdueGL will hand the substructure to *Recursify-Substructure* to see if any of the instances are connected. Since all four of them are connected by an edge, a recursive substructure is created which will cover even more of the input graph, having included three additional edges. Also, it is replaced by a single non-terminal in the input graph, versus four non-terminals when abstracting out the instances non-recursively, one-by-one.

The new productions generated in this iteration are shown in Figure 7. Abstracting away these substructures produces the graph shown in Figure 8.

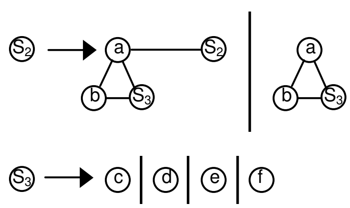



**Figure 7** Second and third productions by SubdueGL

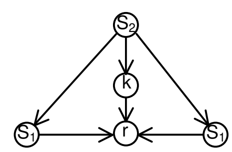



**Figure 8** Input graph parsed by the second and third productions

In the next iteration, SubdueGL cannot find any recurring substructures that can be abstracted out to reduce the graph’s description length. The graph in Figure 8, therefore becomes the right side of the last production. When this rule is executed, the graph is fully parsed.

## Discussion

SubdueGL required only minor extensions to Subdue mainly because of the robustness of the MDL heuristic. Grammars with large amounts of disjunction, either in the form of multiple productions or large discrete ranges for variables, tradeoff with simpler grammars with less coverage. The MDL measure provides a reasonable tradeoff between the two.

The next step in our analysis of the SubdueGL approach is to empirically validate the algorithm and find its limitations. Empirical testing is difficult for graph grammar learning due to the shortage of competitors and real-world examples. However, automated testing may be possible by randomly generating graph grammars, generating examples from these grammars, and then analyze SubdueGL's behavior while attempting to learn the original grammar. Random graph grammar generation is not trivial given the huge space of possible grammars, but such a methodology would allow us some measure of SubdueGL's effectiveness in this domain.

We have also considered a complexity analysis of SubdueGL. Since we are limiting SubdueGL to recursive productions involving only one connecting edge, the complexity is no more than that of Subdue, which is constrained to be polynomial in the size of the input graph.

## Conclusions and Future Work

In this paper we introduced an algorithm, SubdueGL, which is able to learn graph grammars from examples. The algorithm is based on the earlier system called Subdue which has had success in structural data mining for years. SubdueGL focuses on context-free parser graph grammars. Although incomplete, its current capabilities include finding static structures, finding variables, and finding recursive structures.

As mentioned at the beginning, this paper reports on a work in progress. Our future plans include expanding graph grammar learning with concept learning, and handling continuous values. As future results warrant, we may allow variables to take on values that are not restricted to be single vertices. We also plan to investigate other ways to identify recursive structures, with focus on allowing the recursive non-terminal to be embedded in a subgraph, connecting with more than a single edge.

We will evaluate our progress by comparing the system's performance to some competing machine learning algorithms, such as inductive logic programming systems. We also have plans to apply SubdueGL to real-world domains, such as circuit diagrams and protein sequences.

## References


Bartsch-Sörl, B. 1983. Grammatical inference of graph grammars for syntactic pattern recognition. *Lecture Notes in Computer Science*, 153: 1-7.

Carrasco, R. C., J. Oncina, and J. Calera. 1998. Stochastic inference of regular tree languages. *Lecture Notes in Artificial Intelligence*, 1433: 187-198.

Cook, D. J., and L. B. Holder. 2000. Graph-Based Data Mining. *IEEE Intelligent Systems*, 15(2), 32-41.

Cook, D. J., and L. B. Holder. 1994. Substructure Discovery Using Minimum Description Length and Background Knowledge. *Journal of Artificial Intelligence Research*, Volume 1, 231-255.

Engelfriet, J., and G. Rozenberg. 1991. Graph grammars based on node rewriting: an introduction to NLC grammars. *Lecture Notes in Computer Science*, 532, 12-23.

Jeltsch, E., and H. J. Kreowski. 1991. Grammatical inference based on hyperedge replacement. *Lecture Notes in Computer Science*, 532: 461-474.

Nagl, M. 1987. Set theoretic approaches to graph grammars. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Graph Grammars and Their Application to Computer Science*, volume 291 of *Lecture Notes in Computer Science*, 41-54.

Nevill-Manning, C. G., and I. H. Witten. 1997. Identifying Hierarchical Structure in Sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7, 67-82.

Rekers, J., and A. Schürr. 1995. A Parsing Algorithm for Context-Sensitive Graph Grammars. Technical report 95-05, Leiden University.

Rissanen, J. 1989. *Stochastic Complexity in Statistical Inquiry.* World Scientific Company.