MDL-Based Context-Free Graph Grammar Induction

Abstract

We present an algorithm for the inference of context-free graph grammars from examples. The algorithm builds on an earlier system for frequent substructure discovery, and is biased toward grammars that minimize description length. Grammar features include recursion, variables and relationships. We present an illustrative example, demonstrate the algorithm's ability to learn in the presence of noise, and show real-world examples.

Introduction

Acquisition of grammatical knowledge is an important machine learning task with applications in pattern recognition, data mining and computational linguistics. In this research we are concerned with the induction of graph grammars due to the increased expressiveness of graphs over the textual representations typically used for grammar induction. We describe an algorithm for the inference of context-free graph grammars—a set of grammar production rules that describe a graph-based database.

Although textual grammars are useful, they are limited to describing 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.

Grammar induction has a long history. Recent work includes learning string grammars with a bias toward those that minimize description length (Langley and Stromsten 2000), inferring compositional hierarchies from strings in the Sequitur system (Nevill-Manning and Witten 1997), and learning search control from successful parses (Zelle et al. 1994). Only a few algorithms exist for the inference of graph grammars, however. 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 driven by the MDL heuristic (Cook and Holder 2000).

The next section defines the class of context-free graph grammars we are attempting to learn. Next, we describe the graph grammar learning algorithm and give an example. We then present experiments demonstrating the effectiveness of the algorithm on artificial and real-world data. The last section presents conclusions and future work.

Copyright © 2001, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

Context-Free Graph Grammars

We are concerned with graph grammars of the set-theoretic approach, or expression approach (Nagl 1987). Here 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 $S \rightarrow P$, 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 instance of S replaced in the graph. We are interested in learning parse grammars, which applies the production rules in reverse, i.e., replacing an instance of P with S.

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 trivial: 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 $S \rightarrow P\ S$. 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. Alternative productions are of the form $S \rightarrow P_1 \mid P_2 \mid \ldots \mid P_n$, where S is a non-terminal and the P_i are graphs. The non-terminal S can be thought of as a variable having possible values $P_1, P_2, \ldots, P_n$. We will refer to such an S as a variable non-terminal, or simply variable. If P_i are single vertices, then S is synonymous with a regular non-graph discrete variable. It is also possible to generalize such a production to numeric variables by representing possible values via a numeric range: $S \rightarrow [P_{min} \ldots P_{max}]$.

Relationship edges are logical components of the production, in contrast to the structural components: nodes and edges. Therefore, they cannot be a part of the input, but can be part of grammar productions. Relationship edges describe relationships between nodes, such as '*equal to*', and in case of numeric-valued nodes '*less than or equal to*'. See figures 3 and 5 for examples of graph grammar production rules.

Graph Grammar Induction

Our approach to graph grammar induction, called SubdueGL, is based on the Subdue approach (Cook and Holder 2000) for discovering common substructures in graphs. SubdueGL takes data sets in a graph format. The graph representation includes the standard features of graphs: labeled vertices and labeled edges. Edges can be directed or undirected. When converting data to a graph representation, typically objects and values are mapped to vertices, and relationships and attributes are mapped to edges.

Algorithm

The SubdueGL algorithm follows a bottom-up approach to graph grammar learning by performing an iterative search on the input graph such that each iteration results 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 iterates until the entire input graph is abstracted into a single non-terminal, or a user-defined stopping condition is reached.

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. The subgraph definition and all instances are referred to as a substructure. The *ExtendSubstructure* search operator is applied to each of these single-vertex substructures to produce substructures with two vertices and one edge. This operator extends the instances of a substructure by one edge in all possible directions to form new instances. Subsets of similar instances are collected to form new substructures. SubdueGL also considers adding recursion, variables and relations to substructures. These additions are described in separate sections below.

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 S is calculated by Value(S) = DL(S) + DL(G|S), where DL(S) is the description length of the substructure, G is the input graph, and DL(G|S) is the description length of the input graph compressed by the substructure. SubdueGL seeks to minimize the value. Only substructures deemed the best by the MDL principle are kept for further extension.

Recursion

Recursive productions are created by the *RecursifySubstructure* search operator. It is applied to each substructure after the *ExtendSubstructure* operator. *RecursifySubstructure* checks each instance of the substructure to see if it is connected to any of its other instances 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 at the end of an iteration, each such chain of subgraphs is abstracted away and replaced by a single vertex. See figure 3 for an example of a recursive production.

Since SubdueGL discovers commonly occurring substructures first and then attempts to make a recursive production, SubdueGL can only make recursive productions out of lists of substructures that are connected by a single edge, which has 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 cannot yet induce productions such as $S \rightarrow aSb$.

Variables

The first step towards discovering variables is discovering commonly-occurring structures, since variability in the data can be detected by surrounding data. If commonly-occurring structures are connected to vertices with varying labels, these vertices can be turned into variables. See figure 5 for an example of a variable production (S_3).

SubdueGL discovers variables inside the *ExtendSubstructure* search operator. As mentioned before, SubdueGL extends each instance of a substructure in all possible ways and groups the new instances that are alike. After this step, it also groups new instances in a different way. Those that were extended from the same vertex by the same edge in all instances, regardless of what vertex they point to, are grouped together. Let (*v1*, *e*, *v2*) represent edge *e* from vertex *v1* to vertex *v2*. Vertex *v1* is part of the original substructure which was extended by *e* and *v2*. For variable creation, instances are grouped using (*v1*, *e*, *V*), where *V* is a variable (non-terminal) vertex whose values include the labels of all matching *v2*'s. The substructure so created is then evaluated using MDL and competes with others for top placement.

The variable *V* can have many values. It is possible to create other substructures from a subset of these values that may evaluate better according to the MDL principle. Generating all subsets, however, is exponential in the number of unique variable values. We employ a heuristic based on the number of instances in which each unique variable value occurs. The MDL principle prefers values that are supported by many instances. Therefore, new substructures are created by successively removing the value with the lowest support. All these substructures compete for top placement. If all the values of *V* are numeric, then the variable's range is represented using a minimum and maximum value. The above heuristic results in a progressive narrowing of this range.

Relationships

SubdueGL also introduces relationship edges into the grammar. Relationship edges increase the expressive power of a grammar by identifying vertices with labels that are equivalent. In the case of numerical labels the less-than-or-equal-to relationship is also possible via a directed relationship edge.

Relationships are discovered after identifying variables. At least one vertex participating in a relationship has to be a variable non-terminal, since relationships between non-variables are trivial. A relationship edge is identified by comparing a newly discovered variable's values in each instance to every other vertex. If the same relationship holds between the variable and another vertex in every instance of the substructure, a relationship edge is created. Figure 1 shows an example of a production that contains two relationships. The relationship edges are marked with dotted arrows and labeled '<=' and '='.

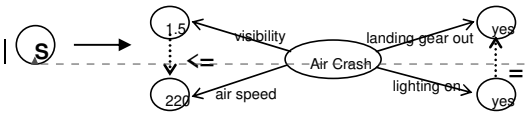

Figure 1. Graph grammar production with relationships.

Example

In this section we give an example of SubdueGL's operation. Consider the input graph shown in figure 2. 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

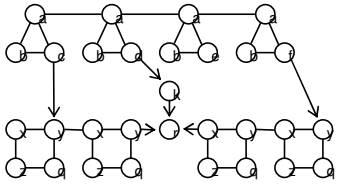

Figure 2. Input graph.

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'.

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 vertex 'x' is expanded 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 first two substructures rank higher, since those have four instances and compress the graph better than the latter two with only 2 and 1 instances respectively.

Applying the *ExtendSubstructure* operator three more times results 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 ranks on the top. Executing the *RecursifySubstructure* operator results in the recursive grammar rule shown in figure 3. 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 2 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 using the above production results in the graph depicted in figure 4.

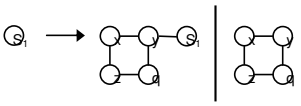

Figure 3. First production generated by SubdueGL.

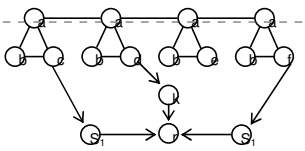

Figure 4. Input graph, parsed by the first production.

The next iteration of SubdueGL uses this graph as its input graph to infer the next grammar rule. Looking at the graph, one can easily see that the most common substructure now is the triangle-looking subgraph. In fact, SubdueGL finds 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 adds different vertices to each instance: 'c', 'd', 'e', and 'f'. The resulting single-instance substructures evaluate poorly by the MDL heuristic.

SubdueGL at this point generates 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 hands the substructure to *RecursifySubstructure* 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 covers 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 5. Abstracting away these substructures produces the graph shown in figure 6.

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 6, therefore becomes the right side of the last production. When this rule is executed, the graph is fully parsed.

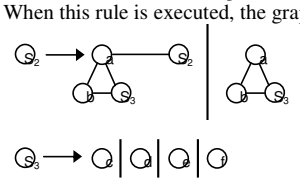

Figure 5. Second and third productions by SubdueGL.

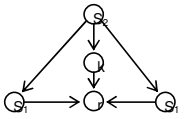

Figure 6. Input parsed by the second and third productions.

Experimental Results

We devised an experiment to show that SubdueGL does not simply discover arbitrary grammars, but finds the most relevant grammars. This proof-of-concept experiment involved using a known graph grammar to generate a graph and having SubdueGL infer the original graph grammar from the graph at varying levels of noise. The grammar we used had nine productions describing graph structures of varying shapes and sizes. Two of these were recursive, and four non-recursive. Three were variables (discrete and continuous) with varying number values. We also had non-terminals on the right side of productions. In other words, we made it quite complex. Due to space restrictions we do not show the grammar here.

The experimental process included the following steps: (1) Generate graph from known grammar, (2) Corrupt the graph by noise, (3) Use SubdueGL to infer a grammar from the corrupted graph, (4) Compute the error by comparing SubdueGL's output with the original grammar. Since the right side of each production is a graph, the error is defined as the number of graph transformations (i.e., insert/delete vertex/edge or change label) needed to transform the known grammar into the inferred grammar.

We corrupted the generated input graph from 0% up to 100% of noise in 5% increments. The noise introduced was defined as the combination of two parameters: the percentage of instances embedded into the graph to be corrupted, and the percentage of a single instance to be corrupted. For example, if we intend to introduce 10% overall noise, we corrupt 31.6% of 31.6% of the instances (31.6% squared being 10%).

Figure 7 shows the results, where the curve represents the average of ten trials. At 0% noise SubdueGL always found the original grammar exactly. We found that in the presence of noise the algorithm has a tendency to add extra values to variables. An argument to SubdueGL that specifies a minimum support for variable values reduces its sensitivity to noise. The minimum support specifies the percentage of instances in which a unique variable value has to appear to be included in a variable production. The results shown in figure 7 were obtained using 10% minimum support, although the results obtained without specifying a minimum support were only slightly worse. As the figure shows, the error stays below 10 transformations, with up to 35% noise.

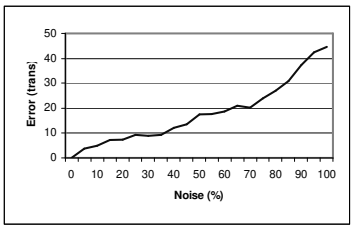

Figure 7. SubdueGL performance on artificial grammar in the presence of noise.

As a real-world example, we applied SubdueGL to protein sequence data. Specifically, we analyzed the primary and secondary structure of the proteins myoglobin and hemoglobin, respectively. These proteins are used widely to illustrate nearly every important feature of protein structure, function, and evolution (Dickerson & Geis, 1983).

S → **S**$_2$ –> **S**$_3$ –> **S**$_4$ –> **S**$_5$ –> **S**$_6$ –> **S**$_7$ –> **S**$_8$ –> **S**$_9$ –> **S**$_{10}$ –> **S**$_{11}$ –> **S**$_{12}$ –> **S**$_{13}$ –> **S**$_{14}$ –> **S**$_{15}$ –> ALA –> **S**

S$_2$ → VAL | LEU | SER | GLU | GLY | TRP | GLN | HIS | ALA | LYS | ASP | ILE | PHE | THR | ASN

S$_3$ → VAL | LEU | SER | GLU | GLN | HIS | ALA | ASP | ILE | ARG | THR | MET | ASN

...

S$_{20}$ → **S**$_{21}$ –> **S**$_{22}$ –> **S**$_{23}$ –> **S**$_{24}$ –> **S**$_{25}$ –> **S**$_{26}$ –> **S**$_{27}$ –> LEU –> **S**$_{20}$

S$_{21}$ → VAL | GLY | GLN | ALA | LYS | ASP | PHE | MET | **S**

S$_{22}$ → VAL | LEU | GLU | GLY | GLN | HIS | ALA | LYS | ILE | PHE | THR | MET | **S**

...

S$_{30}$ → **S**$_{31}$ –> **S**$_{32}$ –> **S**$_{33}$ –> **S**$_{34}$ –> **S**$_{35}$ –> **S**$_{36}$ –> **S**$_{37}$ –> **S**$_{38}$ –> GLY –> **S**$_{30}$

S$_{31}$ → LYS | **S** | **S**$_{10}$

S$_{32}$ → SER | GLU | HIS | LYS | ASP | **S**

S$_{33}$ → VAL | GLU | HIS | ILE | PRO | THR

S$_{34}$ → VAL | GLU | TRP | ALA | ASP | PRO

S$_{35}$ → GLY | ALA | THR

S$_{36}$ → GLY | HIS | LYS | ASP | PHE | PRO

S$_{37}$ → VAL | GLY | LYS | PHE | PRO | TYR

S$_{38}$ → VAL | GLN | HIS | ALA | LYS | THR

S$_{40}$ → **S**$_{30}$ –> **S**$_{30}$ –> **S**$_{20}$ –> **S** –> HIS –> LYS –> LYS –> LYS

Figure 8. Partial grammar induced by SubdueGL on protein primary-sequence data.

The primary structure of myoglobin is represented as a sequence of amino acids, which have a three letter acronym. These compose the vertices of the input graph, which are connected by edges labeled 'next'. The grammar induced by SubdueGL is shown in figure 8, where graph vertices are only shown by their labels. The arrow → is the production operator, while –> signifies the edge 'next' in the graph. For lack of space, we omitted a few variables (**S**$_4$ through **S**$_{15}$, and **S**$_{23}$ through **S**$_{27}$). The expressive power of the grammar is apparent at the first glance. Productions **S**, **S**$_{20}$, and **S**$_{30}$ are recursive, while **S**$_{40}$ contains all these recursive rules followed by a static sequence of amino acids. Rules **S**, **S**$_{20}$, and **S**$_{30}$ each contain a single amino acid at the end of the chain which signifies a recurrence of these amino acids with various combinations of other amino acids in between. Productions **S**$_{21}$, **S**$_{22}$, **S**$_{31}$, and **S**$_{32}$ are also interesting, as they describe regularities among single amino acids, and a recursive sequence of

amino acids. In the case of S_{31}, it can be replaced by LYS, S (a recurrent sequence) or S_{10} (another variable).

For the next example we use the secondary structure of hemoglobin, which is represented in graph form as a sequence of helices and sheets along the primary sequence. Each helix is a vertex which are connected via edges labeled 'next'. Each helix is encoded in the form 'h_t_l', where *h* stands for helix, *t* is the helix type, and *l* is the length. Part of the grammar identified by SubdueGL is shown in figure 9. This grammar only involves helices of type 1 (right-handed α-helix). This grammar can generate the most frequently occurring helix sequences that are unique to hemoglobin. In fact, when compared to the grammars generated for myoglobin and other proteins, the differences can be readily identified.

```
S  → S2 -> S3 -> h_1_6 -> S4 -> h_1_19 -> h_1_8 -> h_1_18 -> S5
S2 → h_1_14 | h_1_15
S3 → h_1_14 | h_1_15
S4 → h_1_6 | h_1_1
S5 → h_1_20 | h_1_23
```

Figure 9. Partial grammar induced by SubdueGL on protein secondary structure data.

Brazma et al. (1998) presented a survey of approaches to automatic pattern discovery in biosequences. Context-free grammars are superior to approaches surveyed there in their ability to represent recursion and relationships among variables.

Conclusions and Future Work

In this paper we introduced an algorithm, SubdueGL, which is able to infer graph grammars from examples. The algorithm is based on the Subdue system which has had success in structural data mining in several domains. SubdueGL focuses on context-free graph grammars. Its current capabilities include finding static structures, finding variables, relationships, recursive structures, and numeric label handling.

We have implemented a concept learning version of SubdueGL as well. It accepts positive and negative input graphs and attempts to construct a grammar that describes the positive graph well, while not describing the negative graph. The value of substructure *S* is computed as $\text{Value}(S) = \text{DL}(S) + \text{DL}(G^+|S) + \text{DL}(G^-) - \text{DL}(G^-|S)$, where DL(S) is the description length of the substructure, G^+ is the positive input graph, G^- is the negative input graph. We are currently conducting experiments using concept learning.

Despite the advantages of SubdueGL's expressive power, there is room for improvement. As mentioned before, recursive productions can only be formed out of recurring sequences using a single edge. At this point, variable productions can only have single vertices on the right side of the production. Even though the vertex can be a non-terminal, there might be advantages to allowing arbitrary graphs as well.

Our experiments show that the algorithm is somewhat susceptible to noise when forming variable productions. Specifying a minimum support alleviates most of this problem, but human judgment is needed in specifying the level of support.

Our future plans include work on second-order graph grammar inference, where preliminary results show promise. We may also consider work on probabilistic graph grammars. 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 also plan to compare our approach to ILP and other competing systems.

Acknowledgments

This research is partially supported by a grant from the Defense Advanced Research Projects Agency and managed by Rome Laboratory under contract F30602-01-2-0570.

References

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

Brazma, A., I. Jonassen, I. Eidhammer, D. Gilbert. 1998. Approaches to automatic discovery of patterns in biosequences. *Journal of Computational Biology*, Vol. 5, Nr. 2, 277-303.

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

Dickerson, R.E. and I. Geis. 1982. Hemoglobin: structure, function, evolution, and pathology. Benjamin/Cummings Inc.

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.

Langley, P. and Stromsten, S. 2000. Learning context-free grammars with a simplicity bias. Proceedings of the *Eleventh European Conference on Machine Learning,* 220-228. Barcelona: Springer-Verlag.

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.

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

Zelle, J. M., R. J. Mooney, and J. B. Konvisser. 1994. Combining top-down and bottom-up methods in inductive logic programming. Proceedings of the *Eleventh International Conference on Machine Learning*, 343-351.