International Journal on Artificial Intelligence Tools
Vol. 13, No. 1 (2004) 65–79
© World Scientific Publishing Company

MDL-BASED CONTEXT-FREE GRAPH GRAMMAR INDUCTION AND APPLICATIONS

ISTVAN JONYER
Department of Computer Science
Oklahoma State University
700 Greenwood, Tulsa, OK 74107
jonyer@cs.okstate.edu

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
{holder,cook}@cse.uta.edu

Received 18 July 2003
Revised 19 October 2003
Accepted 19 October 2003

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.

Keywords: Grammar induction; graph-based data mining; machine learning.

1. 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 expressive power 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 and object-oriented 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[1] inferring compositional hierarchies from strings in the Sequitur system[2], and learning search control from successful parses[3]. 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[4]. Other algorithms utilize a merging technique for hyperedge replacement grammars[5] and regular tree grammars[6]. Our approach is based on a method for discovering frequent substructures in graphs driven by the MDL heuristic[7].

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.

2. Context-Free Graph Grammars

When learning a grammar in general, one has to decide the intended use of the grammar. Grammars come in two flavors: those that parse and those that generate languages. Parser grammars are optimized for fast parsing, sacrificing some accuracy which results in grammars that over-accept. That is, they accept sentences that are not in the language. Generator grammars trade accuracy for speed as well. As expected, they are not able to generate the entire language. Grammars that can generate and parse the same language exactly are very hard to design and are usually too big and slow to be practical. In our work, we are proposing a third use: knowledge discovery. Here, we do not generate grammars to be used in practical and efficient parser or generator algorithms. Instead, we are simply interested in exploiting the fact that grammars are able to generalize beyond the examples used to infer them.

In this research we are addressing the problem of inferring graph grammars from examples. In other words, we seek to design a machine learning algorithm in which the learned hypothesis is a graph grammar.

Machine learning algorithms in general attempt to learn theories that generalize beyond the seen examples, such that new, unseen data can be accurately categorized. Translated to grammar terms, we would like to learn grammars that accept 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.

We are concerned with graph grammars of the set-theoretic approach, or expression approach[8]. 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 that connect P 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[9]. 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. (Terminal vertices are those with actual vertices and edges, while non-terminals are like variables.) 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, ..., 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.

3. Graph Grammar Induction

Our approach to graph grammar induction, called SubdueGL, is based on the Subdue approach[7] 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, objects and values are mapped to vertices, and relationships and attributes are mapped to edges.

3.1. *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[10], and applied to graph-based knowledge discovery by Cook and Holder[11]. 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.

3.1.1. *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.

3.1.2. *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*, also expressing the direction of the directed edge *e*. 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.

3.1.3. *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 '='.

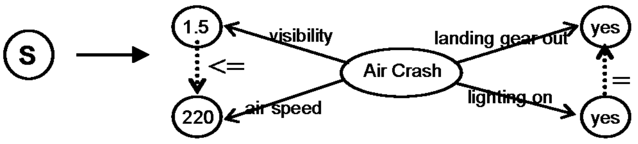

Figure 1. Graph grammar production with relationships.

3.1.4. *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 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’.

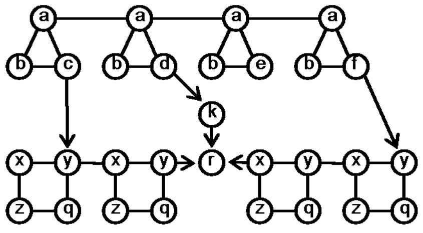

Figure 2. Input graph.

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 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.

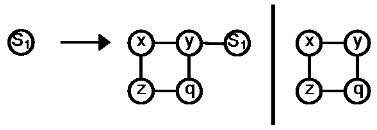

Figure 3. First production generated by SubdueGL.

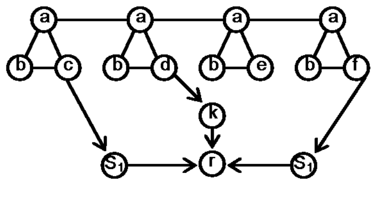

Figure 4. Input graph, parsed by the first production.

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.

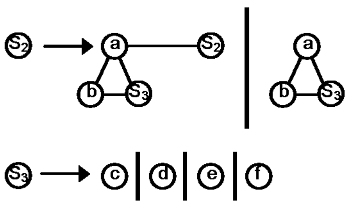

Figure 5. Second and third productions by SubdueGL.

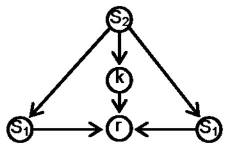

Figure 6. Input parsed by the second and third productions.

4. Experimental Results

This section describes the results of experiment ran using the above described algorithm. Empirical evaluation is described first, followed by real-world examples.

4.1. *Empirical Evaluation*

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 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.

The experiments used three graph grammar rules that we will call the *triangle*, the *star* and the *complex* rules. These are shown in figure 7. The edges in the triangle and star productions are labeled uniformly, and in the complex production, non-uniformly.

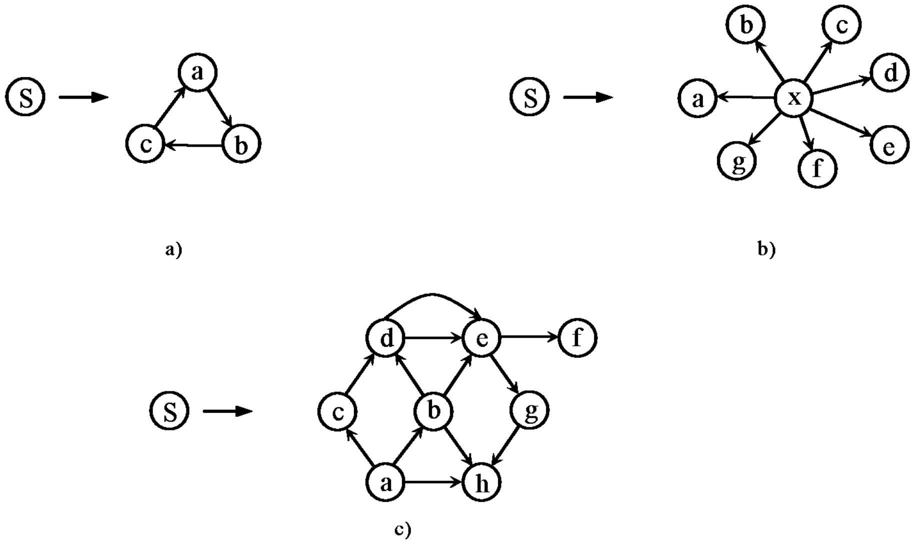

Figure 7. a) Triangle production b) Star production c) Complex production

We use these three production rules as the basis for our experiments. They represent an increasingly complex set of rules that SubdueGL will have to deal with. The triangle rule is, of course, the simplest one, being a simple closed shape with three vertices and three edges. With this shape we would like to determine SubdueGL's ability to find small patterns.

The star pattern is frequently used in the graph representation of flat (non-relational) domains, therefore it is important to know how SubdueGL deals with this type of pattern. The star we use simulates the presence of seven attributes, each corresponding to each edge of the star.

The complex shape was designed to include most basic graph features to a certain complexity. It includes closed triangle shapes, squares with varying edge directions, pentagons, multiple edges between vertices, and a vertex that is attached to the rest of the graph with only a single edge.

These basic shapes were extended to include recursion and variables. Recursive edges were added to each rule, which are used to connect instances after the instances are generated. These are typically labeled 'next'. The same way, discrete and continuous variables were also added. We conducted a large number of experiments using all combination of the features, the basic shapes alone, using recursion, using variables, and using both. We also combined all these manually generated rules to create even more complex graph grammars.

Another concern about grammar rules is their number of occurrence in the graph generated. We would like to find out how the number of embedded instances affects SubdueGL's ability to discover them. Therefore, we will not predetermine a fixed number, but rather run experiments using varying number of instances to experimentally determine their effect.

We corrupted the generated input graph by deleting edges and vertices, and re-labeling (introducing label noise) both vertices and edge. We added noise from 0% up to 100% 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 8 shows the error plots for a specific experiment, but these results are quite typical. The five error curves represent trials using different number of embeddings of each rule. We used 2, 4, 8, 16, and 32 instances. The curves represent the average of thirty trials, which was sufficient for producing accurate results with a confidence interval of 1.0 error with 95% confidence. In other words, there was a 5% chance that the error was greater than 1.0 transformations. In some cases, however, many more runs would have been necessary to obtain the same confidence. This number could be as high as 200 runs for certain experiments. Since the number of runs necessary varies for each experiment, it was infeasible to compute this number for each experiment in a reasonable amount of time. Instead, the number of runs necessary were computed only for the experiments included in figure 8. For the remaining experiments thirty runs were used, which are actually good enough to give an idea of the shape of the curves. More runs would have made the curves look smoother, but for our purposes they sufficiently depict the performance behavior.

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.

Results from 2-instance experiments easily explained that with no noise, SubdueGL will always find the original grammar, but with one instance corrupted, most of the regularity is gone. Therefore, it becomes impossible for SubdueGL to find the original grammar. A single small substructure can still be found close to 25% noise, where one of the instances (50%) is corrupted by 50% (the total noise being $50\%^2 = 25\%$). For a large enough grammar rule size the other 50% of the noisy instance that is left intact may still be found, having 2 instances (one in the noisy instance and one in the noise-free instance). All of this 50%, however, would have to be connected, such that a connected substructure can be found. The probability of this is low. And even when it does happen, the error associated with a rule found in these circumstances is still very high. This is the reason why we see the 2-instance experiments go from zero error to very close to the maximum error very fast, at 25% noise.

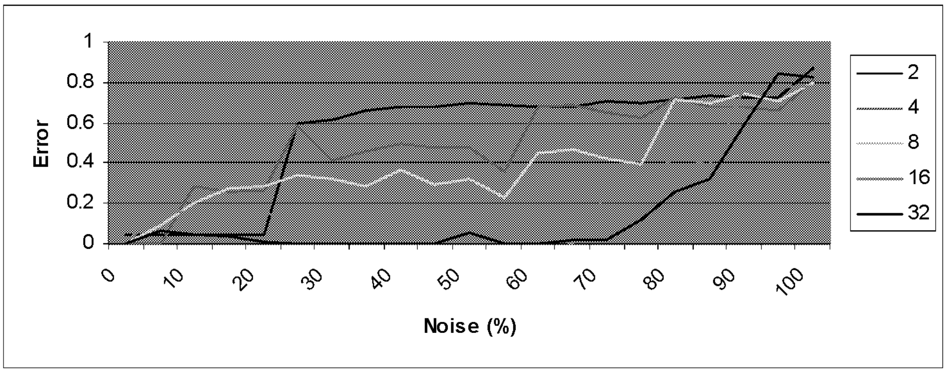

Figure 8. Typical error curves on artificial domains by SubdueGL

We have explained the shape of the error curve associated with the 2-instance experiments. Looking at the experiments with more instances, the general shape of the curves emerge, although their values change relative to each other. Figure 9 gives a schematic view of the general error curve, where significant features are also marked. As we can see, the curve is not a monotonically increasing or decreasing one. It has local minima and maxima. We show four of these features in the figure, but in reality there may be any number of these. The typical curve can be imagined as a stair-like curve with varying number of steps. The closest to the general curve is the one for 8-instances in figure 8. On the other curves, these features are less pronounced, because these local extremes are exaggerated, seemingly taking over the overall shape of the curve.

The local extremes are due to tradeoffs that happen in the learned grammar while SubdueGL is attempting to minimize the description length. To review, the MDL heuristic provides a tradeoff between substructure frequency and size. If SubdueGL was only to look for frequent substructures, it would terminate at single-vertex substructures, as adding to a substructure will most likely reduce its frequency. The MDL heuristic rewards for increased substructure size, driving the search towards a substructure that best trades off size and frequency.

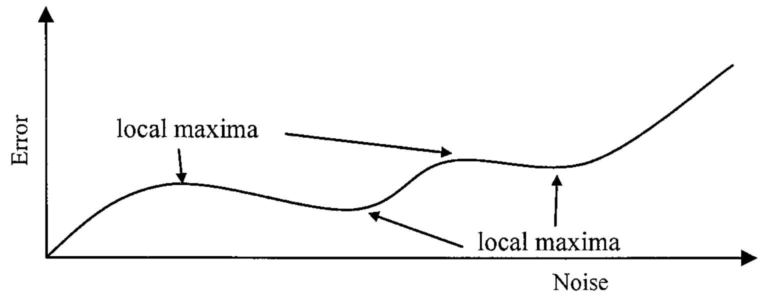

Figure 9. General error curve

Let us look at what is happening in the typical error curve in figure 8. Let us start from 0% noise, where SubdueGL recovers the original grammar exactly. Introducing a small amount of error corrupts a small number of instances in a small amount. Such error prompts SubdueGL to prefer slightly smaller substructures in exchange for the added instances. The more SubdueGL gives up of the substructure in exchange for more instances, the more error it will incur when the learned grammar is compared to the original one.

As more noise is introduced, the noise starts tipping the balance of the tradeoff towards favoring larger substructures, which are closer to the original grammar. The error starts to decrease until the balance is tipped again. This happens when the number of noiseless instances are in rare supply and SubdueGL finds more frequent patterns that are smaller in size. Given enough instances and a large enough substructure to begin with, the balance can tip many times to form a stair-like error curve until 100% noise is introduced and the maximum error is incurred. Figure 10 depicts a scenario in which SubdueGL must determine whether to favor a smaller, more frequent substructure over a larger, less frequent one. Learned patterns in both figure 10a and figure 10b are drawn in bold lines. In figure 10a, there are two instances of a pattern that has a total size of 19 vertices and edges, for a total coverage of 38 vertices and edges in the input graph. In figure 10b, there are four instances of a pattern that has 10 vertices and edges, for a total coverage of 40. Although the description length calculation is more involved, this gives us a good idea of the type of tradeoff SubdueGL has to make in terms of size and frequency.

If error was introduced in a different way such that a small number of instances were corrupted by a large percentage, SubdueGL would not be able to make the tradeoff of smaller structures for more instances. The error curve would be modified such that no error would be incurred for a small amount of noise. In fact, if many of the noisy instances were corrupted, SubdueGL would recover the original grammar accurately even with a large amount of noise. In other words, looking at the 32-instance experiment in figure 8, the curve would remain flat up to about 70% noise.

Additionally, more instances can be added if they are structurally identical to the other instances and differ only in vertex labels. In those cases variable productions are created.

If the vertex was originally not a variable, this tactic for more instances leaves SubdueGL with lower accuracy.

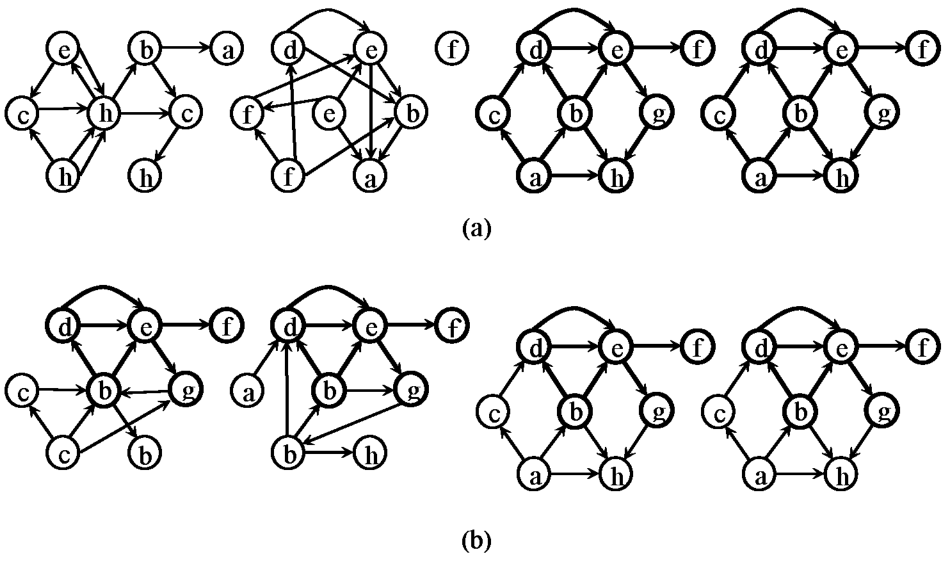

Figure 10. a) Tradeoff 1: few large patterns b) Tradeoff 2: many small patterns

Looking at the error curves, we may notice that the actual error does not reach 100%. This happens because vertex labels that are to be corrupted by noise are replaced with labels from the pool of already available labels. Even though its original label is not assigned to any vertex, the label that is assigned to a vertex comes from the original specification of the grammar. Any grammar rule found will have vertices in it with labels that will match the original grammar. At least these vertices will not have to be changed, and that is a deduction from the match cost of the induced and the original grammars. The smaller the rule the more likely that edges are reconnected to vertices in a way that is found in the original grammar.

We may notice a sudden jump in the error curves at 100% noise. The number of instances unaffected by noise at 100% is zero. Even though typically only one of the instances is noiseless at 95% noise, it gives a small pattern a chance to be found elsewhere in the input graph, that was left unaffected by noise elsewhere in the graph. At 100% noise everything is randomized and the smallest patterns disappear as well.

4.2. *Real-world experiments*

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[12].

$S \rightarrow$ 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
$S_2 \rightarrow$ VAL | LEU | SER | GLU | GLY | TRP | GLN | HIS |
ALA | LYS | ASP | ILE | PHE | THR | ASN
$S_3 \rightarrow$ VAL | LEU | SER | GLU | GLN | HIS | ALA | ASP | ILE | ARG | THR | MET | ASN
. . .
$S_{20} \rightarrow$ S_{21} -> S_{22} -> S_{23} -> S_{24} -> S_{25} -> S_{26} -> S_{27} -> LEU -> S_{20}
$S_{21} \rightarrow$ VAL | GLY | GLN | ALA | LYS | ASP | PHE | MET | S
$S_{22} \rightarrow$ VAL | LEU | GLU | GLY | GLN | HIS | ALA | LYS |ILE | PHE | THR | MET | S
. . .
$S_{30} \rightarrow$ S_{31} -> S_{32} -> S_{33} -> S_{34} -> S_{35} -> S_{36} -> S_{37} -> S_{38} -> GLY -> S_{30}
$S_{31} \rightarrow$ LYS | S | S_{10}
$S_{32} \rightarrow$ SER | GLU | HIS | LYS | ASP | S
$S_{33} \rightarrow$ VAL | GLU | HIS | ILE | PRO | THR
$S_{34} \rightarrow$ VAL | GLU | TRP | ALA | ASP | PRO
$S_{35} \rightarrow$ GLY | ALA | THR
$S_{36} \rightarrow$ GLY | HIS | LYS | ASP | PHE | PRO
$S_{37} \rightarrow$ VAL | GLY | LYS | PHE | PRO | TYR
$S_{38} \rightarrow$ VAL | GLN | HIS | ALA | LYS | THR
$S_{40} \rightarrow$ S_{30} -> S_{30} -> S_{20} -> S -> HIS -> LYS -> LYS -> LYS

Figure 11. 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 11, where graph vertices are only shown by their labels. The arrow $\rightarrow$ 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 the above mentioned proteins, which is represented in graph form as a sequence of helices and sheets along the primary sequence. Each helix is a vertex which is 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 12. 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. It can also generate others as well, for instance, ones that also occur in myoglobin. The grammar generated for myoglobin secondary sequence is shown in figure 13. We can see that the two sequences have the same length and the languages generated by them intersect in the sequence

h_1_15 –> h_1_15 –> h_1_6 –> h_1_6 –> h_1_19 –> h_1_8 –> h_1_18 –> h_1_23.

This is interesting, since researchers have long been speculating that a common evolutionary path exists for these proteins. Hou and colleagues[13] reported on myoglobin-like proteins that are prime candidates to be common ancestors.

$S \rightarrow S_2$ -> S_3 -> h_1_6 -> S_4 -> h_1_19 -> h_1_8 -> h_1_18 -> S_5
$S_2 \rightarrow$ h_1_14 | h_1_15
$S_3 \rightarrow$ h_1_14 | h_1_15
$S_4 \rightarrow$ h_1_6 | h_1_1
$S_5 \rightarrow$ h_1_20 | h_1_23

Figure 12. Partial grammar by SubdueGL on hemoglobin secondary structure.

$S \rightarrow$ h_1_15 -> h_1_15 -> h_1_6 -> h_1_6 -> h_1_19 -> S_2 -> h_1_18 -> S_3
$S_2 \rightarrow$ h_1_9 | h_1_8
$S_3 \rightarrow$ h_1_25 | h_1_23

Figure 13. Partial grammar by SubdueGL on myoglobin secondary structure.

Brazma and colleagues[14] 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.

5. 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.

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 can alleviate this problem, but human judgment is needed in specifying the 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.

6. Acknowledgments

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

7. Referencses

[1] 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.

[2] 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.

[3] 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 ICML*, 343-351.

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

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

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

[7] Cook, D.J. and L.B. Holder. 2000. Graph-based data mining. *IEEE Intelligent Systems*, 15(2), 32-41.

[8] 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.

[9] 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.

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

[11] 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

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

[13] Hou, S., R. W. Larsen, D. Boudko, C. W. Riley, E. Karatan, M. Zimmer, G. W. Ordal And M. Alam. 2000. *Myoglobin-like aerotaxis transducers in Archaea and Bacteria.* Nature 403, 540–544.

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