Inference of Node Replacement Graph Grammars

Jacek P. Kukluk, Lawrence B. Holder, and Diane J. Cook

Contact Author:

Lawrence B. Holder HOLDER @CSE.UTA.EDU
Department of Computer Science and Engineering
University of Texas at Arlington
Box 19015, Arlington, TX 76019
Office: (817) 272-2596
Fax: (817) 272-3784

Running Title:

Inference of Node Replacement Graph Grammars

Abstract

Graph grammars combine the relational aspect of graphs with the iterative and recursive aspects of string grammars, and thus represent an important next step in our ability to discover knowledge from data. In this paper we describe an approach to learning node replacement graph grammars. This approach is based on previous research in frequent isomorphic subgraphs discovery. We extend the search for frequent subgraphs by checking for overlap among the instances of the subgraphs in the input graph. If subgraphs overlap by one node we propose a node replacement grammar production. We also can infer a hierarchy of productions by compressing portions of a graph described by a production and then infer new productions on the compressed graph. We validate this approach in experiments where we generate graphs from known grammars and measure how well our system infers the original grammar from the generated graph. We also describe results on several real-world tasks from chemical mining to XML schema induction. We briefly discuss other grammar inference systems indicating that our study extends classes of learnable graph grammars.

Keywords: Grammar Induction, Graph Grammars, Graph Mining.

Inference of Node Replacement Recursive Graph Grammars

Jacek P. Kukluk, Lawrence B. Holder, and Diane J. Cook

Jacek P. Kukluk KUKLUK@CSE.UTA.EDU
Lawrence B. Holder HOLDER @CSE.UTA.EDU
Diane J. Cook COOK @CSE.UTA.EDU
Department of Computer Science and Engineering
University of Texas at Arlington
Box 19015, Arlington, TX 76019

Editor:

Abstract

Graph grammars combine the relational aspect of graphs with the iterative and recursive aspects of string grammars, and thus represent an important next step in our ability to discover knowledge from data. In this paper we describe an approach to learning node replacement graph grammars. This approach is based on previous research in frequent isomorphic subgraphs discovery. We extend the search for frequent subgraphs by checking for overlap among the instances of the subgraphs in the input graph. If subgraphs overlap by one node we propose a node replacement grammar production. We also can infer a hierarchy of productions by compressing portions of a graph described by a production and then infer new productions on the compressed graph. We validate this approach in experiments where we generate graphs from known grammars and measure how well our system infers the original grammar from the generated graph. We also describe results on several real-world tasks from chemical mining to XML schema induction. We briefly discuss other grammar inference systems indicating that our study extends classes of learnable graph grammars.

Keywords: Grammar Induction, Graph Grammars, Graph Mining.

1. Introduction

String grammars are fundamental to linguistics and computer science. Graph grammars can represent relations in data which strings cannot. Graph grammars can represent hierarchical structures in data and generalize knowledge in graph domains. They have been applied as analytical tools in physics, biology, and engineering [7][15]. Our objective with this research is to develop algorithms for graph grammar inference capable of learning recursive productions. We introduce an algorithm which builds on previous work in discovering frequent subgraphs in a graph [5]. We check if subgraphs overlap and if they overlap by one node, we use this node and subgraph structure to propose a node replacement graph grammar. We also can infer a hierarchy of productions by compressing portions of a graph described by a production and then infer new productions on the compressed graph. We validate this approach in experiments where we generate graphs from known grammars and measure how well our system infers the original grammar from the generated graph. We show how inference error depends on noise, complexity of the grammar, number of labels, and size of input graph. We also describe results on several real-world tasks from chemical mining to XML schema induction. A vast amount of research has been done in string grammar inference [20]. We found only a few studies in graph grammar inference, which we describe in the next section.

In the remainder of the paper we the first define graph grammars. Next we describe our algorithm for inferring graph grammars. Then, we show experiments to reveal the advantages and limitations of our method. We close with conclusions and future directions.

2. Related work

Jeltsch and Kreowski [9] did a theoretical study of inferring hyperedge replacement graph grammars from simple undirected, unlabeled graphs. Their paper leads through an example where from four complete bipartite graphs $K_{3,1}$, $K_{3,2}$, $K_{3,3}$, $K_{3,4}$, the authors describe the inference of a grammar that can generate a more general class of bipartite graphs $K_{3,n}$, where $n \geq 1$. The authors define four operations that lead to a final hyperedge replacement grammar. The operations are: INIT, DECOMPOSE, RENAME, and REDUCE. The INIT operation will start the process from a grammar which has all sample graphs in its productions and therefore it generates only the sample graphs. Then, the DECOMPOSE operation transforms the initial productions into productions that are more general but can still produce every graph from the sample graphs. RENAME allows for changing names of non-terminal labels. REDUCE removes redundant productions. Jeltsch and Kreowski [9] start the process from a grammar which has all the sample graphs in its productions. Then they transform the initial productions into productions that are more general but can still produce every graph from the sample graphs. Their approach guarantees that the final grammar will generate graphs that contain all sample graphs.

Oates et al. [17] discuss the problem of inferring probabilities of every grammar rule for stochastic hyperedge replacement context free graph grammars. They call their program Parameter Estimation for Graph Grammars (PEGG). They assume that the grammar is given. Given a structure of a grammar S and a finite set of graphs E generated by grammar S, they ask what are the probabilities θ associated with every rule of the grammar. Their strategy is to look for a set of parameters θ that maximizes the probability $p(E| S, \theta)$.

In terms of similarity to string grammar inference we consider the Sequitur system developed by Nevill-Manning and Witten [16]. Sequitur infers hierarchical structure by replacing substrings based on grammar rules. The new, compressed string is searched for substrings which can be described by grammar rules, and they are then compressed with the grammar and the process continues iteratively. Similarly, in our approach we replace the part of a graph described by the inferred graph grammar with a single node and we look for grammar rules on the compressed graph and repeat this process iteratively until the graph is fully compressed.

The most relevant work to this research is Jonyer et al.'s approach to node-replacement graph grammar inference [10][11]. Their system starts by finding frequently occurring subgraphs in the input graphs. Frequent subgraphs are those that when replaced by single nodes minimize the description length of the graph. They check if isomorphic instances of the subgraphs that minimize the measure are connected by one edge. If they are, a production $S \rightarrow PS$ is proposed, where P is the frequent subgraph. P and S are connected by one edge. Our approach is similar to Jonyer's in that we also start by finding frequently occurring subgraphs, but we test if the instances of the subgraphs overlap by one node. Jonyer's method of testing if subgraphs are adjacent by one edge limits his grammars to description of "chains" of isomorphic subgraphs connected by one edge. Since an edge of a frequent subgraph connecting it to the other isomorphic subgraph can be included to the subgraph structure, testing subgraphs for overlap allows us to propose a class of grammars that have more expressive power than the graph structures covered by Jonyer's grammars. For example, testing for overlap allows us to propose grammars which can describe tree structures, while Jonyer's approach does not allow for tree grammars. In Figure 1 we illustrate the limitation of Jonyer's approach. We generated a tree from a grammar that has two productions. The first production is selected 60% of a time, and the second production is a terminal which is a single vertex and is selected 40% of a time. The grammar found by Jonyer's approach, and resulting compressed graph, are shown on the right side of the figure. The grammar represents the

chain of patterns connected by one edge in the graph, but it cannot represent the tree. The approach we propose in this paper does not have this limitation.

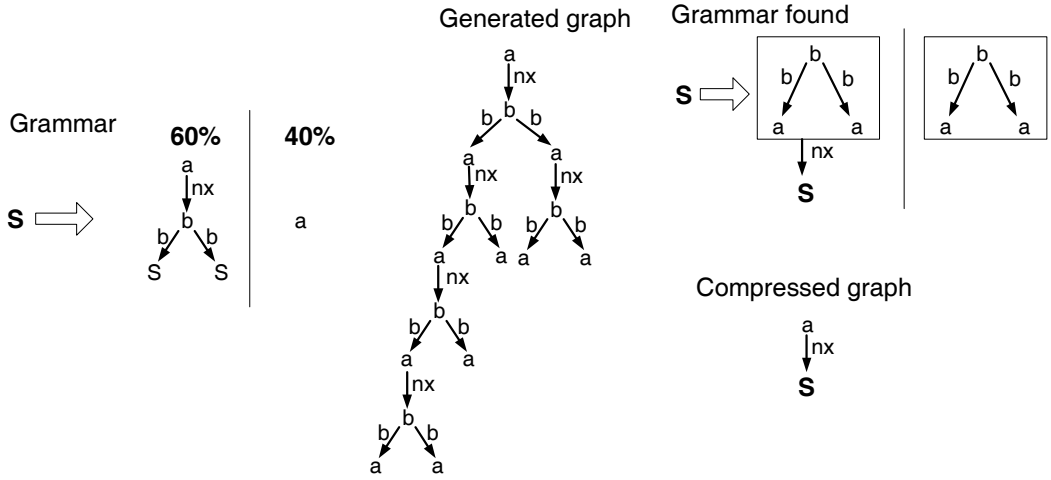


Figure 1. Jonyer's approach to graph grammar inference finds chain of patterns in the tree.

In our approach we use the frequent subgraph discovery system Subdue developed by Cook and Holder [5]. We would like to mention other systems developed to discover frequent subgraphs and therefore have the potential to be modified into a system which can infer a graph grammar. Kuramochi and Karypis [13] implemented the FSG system for finding all frequent subgraphs in large graph databases. FSG starts by all frequent one and two edge subgraphs. Then, in each iteration, it generates candidate subgraphs by expanding the subgraphs found in the previous iteration by one edge. In every iteration, the algorithm checks how many times the candidate subgraph occurs within an entire graph. The candidates, whose frequency is below a user-defined level, are pruned. The algorithm returns all subgraphs occurring more frequently than the given level. In the candidate generation phase, computation costs of testing graphs for isomorphism are reduced by building a unique code for the graph (canonical labeling).

Yan and Han introduced gSpan [24], which does not require candidate generation to discover frequent substructures. The authors combine depth first search and lexicographic order in their algorithm. Their algorithm starts from all frequent one-edge graphs. The labels on these edges together with labels on incident nodes define a code for every such graph. Expansion of these one-edge graphs maps them to longer codes. The codes are stored in a tree structure such that if $\alpha = (a_0, a_1, \ldots, a_m)$ and $\beta = (a_0, a_1, \ldots, a_m, b)$, then the β code is a child of the α code. Since every graph can map to many codes, the codes in the tree structure are not unique. If there are two codes in the code tree that map to the same graph and one is smaller then the other, the branch with the smaller code is pruned during depth first search traversal of the code tree. Only the minimum code uniquely defines the graph. Code ordering and pruning reduces the cost of matching frequent subgraphs in gSpan.

Since gSpan and FSG are both frequent subgraph mining systems, they also can be used to infer graph grammars with the modification to allow subgraph instances to overlap and a mechanism for overlap detection. Modification of gSpan and FSG where the graph would be compressed with frequent isomorphic subgraphs would allow for building a hierarchy of productions. Overlapping substructures are available as an option in the Subdue system [5]. Also, Subdue allows for identification of one substructure with the best compression score, which we can modify to identify one grammar production with the best score, while FSG and gSpan return all candidate subgraphs above a user-defined frequency level leaving interpretation and final selection for the user.

3. Node replacement recursive graph grammar

We give the definition of a graph and graph grammars which is relevant to our implementation. The defined graph has labels on vertices and edges. Every edge of the graph can be directed or undirected. We emphasize the role of recursive productions in the name of the grammar, because the type of inferred productions are such that the non-terminal label on the left side of the production appears one or more times in the node labels of a graph on the right side. It is the main characteristic of our grammar productions. Our approach can also infer non-recursive productions. The embedding mechanism of the grammar consists of connection instructions. Every connection instruction is a pair of vertices that indicate where the production graph can connect to itself in a recursive fashion. Our graph generator can generate a larger class of graph grammars than defined below. We will describe the grammars used in generation later in the paper.

A *labeled graph* G is a 6-tuple, $G = (V, E, \mu, \nu, \eta, L)$, where
V - is the set of nodes,
$E \subseteq V \times V$ - is the set of edges,
$\mu : V \to L$ - is a function assigning labels to the nodes,
$\nu : E \to L$ - is a function assigning labels to the edges,
$\eta : E \to \{0,1\}$ - is a function assigning direction property to edges (0 if undirected, 1 if directed).
L - is a set of labels on nodes and edges.

A *node replacement recursive graph grammar* is a tuple $Gr = (\sum, \Delta, \Gamma, P)$, where
$\sum$ - is an alphabet of node labels,
Δ - is an alphabet of terminal node labels, $\Delta \subseteq \sum$,
Γ - is an alphabet of edge labels, which are all terminals,
P - is a finite set of productions of the form (d, G, C), where $d \in \sum - \Delta$, G is a graph, and there are two types of productions:
(1) *recursive productions* of the form (d, G, C), with $d \in \sum - \Delta$, G is a graph, where there is at least one node in G labeled d. C is an embedding mechanism with a set of connection instructions, $C \subseteq V \times V$, where V is the set of nodes of G. A connection instruction $(v_i, v_j) \in C$ implies that derivation can take place by replacing v_i in one instance of G with v_j in another instance of G. All the edges incident to v_i are incident to v_j. All the edges incident to v_j remain unchanged.
(2) *non-recursive production,* there is no node in G labeled d (our inference system does not infer an embedding mechanism for these productions).

Each grammar production can have one or more connection instructions. If the grammar production does not have a connection instruction, it is a non-recursive production. Each connection instruction consists of two integers. They are the numbers of vertices in two isomorphic subgraphs. In Figure 4 we show three isomorphic instances and connection instructions. Connection instructions are determined from overlap. They show how instances overlap in the input graph and can be used in generation. We compress portions of the graph described by productions. Connection instructions show how one instance connects to its isomorphic copy. They do not show how an instance is connected to the compressed graph. We intended to give a definition of a graph grammar which describes the class of grammars which can be inferred by our approach. We do not infer the embedding mechanism of recursive and non-recursive productions for the compressed graph, but this is an issue for further theoretical and experimental study. When a production is non-recursive, instances do not overlap and do not connect to each other. We do not explicitly give an embedding mechanism for this case.

We introduce the definition of two data structures used in our algorithm.

Substructure S of a graph *G* is a data structure which consists of:

(1) graph definition of a substructure S_G which is a graph isomorphic to a subgraph of *G*

(2) list of instances ($I_1, I_2, \ldots, I_n$) where every instance is a subgraph of *G* isomorphic to S_G

Recursive substructure recursiveSub is a data structure which consists of:

(1) graph definition of a substructure S_G which is a graph isomorphic to a subgraph of *G*

(2) list of connection instructions which are pairs of integer numbers describing how instances of the substructure can overlap to comprise one instance of the corresponding grammar production rule.

(3) list of recursive instances $(IR_1, IR_2, \ldots, IR_n)$ where every instance IR_k is a subgraph of *G*. Every instance IR_k consist of one or more isomorphic, overlapping by no more than one vertex, copies of S_G.

In our definition of substructure we refer to subgraph isomorphism. However, in our algorithm we are not solving the subgraph isomorphism problem. We are using a polynomial time beam search to discover substructures and graph isomorphism to collect instances of the substructures.

4. Hierarchy of graph grammars

We encountered in existing literature a classification of graph grammars based on the embedding mechanism [12]. The embedding mechanism is important in the generation process, but if we use graph grammars in parsing or as a tool to mine data and visualize common patterns, the embedding mechanism may have less importance or can be omitted. Without the embedding mechanism the graph grammar still conveys information about graph structures used in productions and relations between them. In Figure 2 we give the classification of graph grammars based on the type of their productions, not based on the type of embedding mechanism. The production of the grammars in the hierarchy is of the form (d, G, C) where *d* is the left hand side of the production, *G* is a graph, and *C* is the embedding mechanism. *d* can be a single node, a single edge or a graph, and we respectively call the grammar a node-, edge- or graph replacement grammar. If the replacement of *d* with *G* does not depend on vertices adjacent to *d* or edges incident to *d,* nor any other vertices or edges outside *d* in a graph hosting *d*, we call the grammar context free. Otherwise, the grammar is context sensitive.

We wanted to place the graph grammars we are able to infer in this hierarchy. We circled two of them. *Node replacement recursive graph grammar* is the one described in this paper. The set of grammars inferred by Jonyer et al. [10][11] we call c*hain grammars*. Chain grammars describe graphs or portion of graphs composed from isomorphic subgraphs where every subgraph is adjacent to the other by one edge. The productions of *chain grammars* are of the form $S \rightarrow PS$, where *P* is the subgraph. *P* and *S* are connected by one edge. *Chain grammars* are a subset of *node replacement recursive graph grammars. Node replacement graph grammars* describe a more general class of graph grammars than our algorithm is able to learn. An example of a node replacement graph grammar that we cannot learn is a grammar with alternating productions, as in Figure 15.

5. The algorithm

We will first describe an algorithm informally allowing for an intuitive understanding of the idea. The example in Figure 3 shows a graph composed of three overlapping substructures. The algorithm generates candidate substructures and evaluates them using the following measure of compression,

$$\frac{size(G)}{size(S)+size(G \mid S)}$$

where G is the input graph, S is a substructure and $G \mid S$ is the graph derived from G by compressing each instance of S into a single node. $size(g)$ can be computed simply by adding the number of nodes and edges: $size(g) = vertices(g) + edges(g)$. Another successful measure of $size(g)$ is the Minimum Description Length (MDL) discussed in detail in [19][4]. Either of these measures can be used to guide the search and determine the best graph grammar. In our experiments we used only the size measure. The compression measure could also take into account the connection instructions. We could penalize recursive substructures for the additional information connection instruction carry. We did preliminary experiments incorporating connection instruction in the measure, but we did not observe significant improvements in results and therefore we decided to leave the measure unchanged.

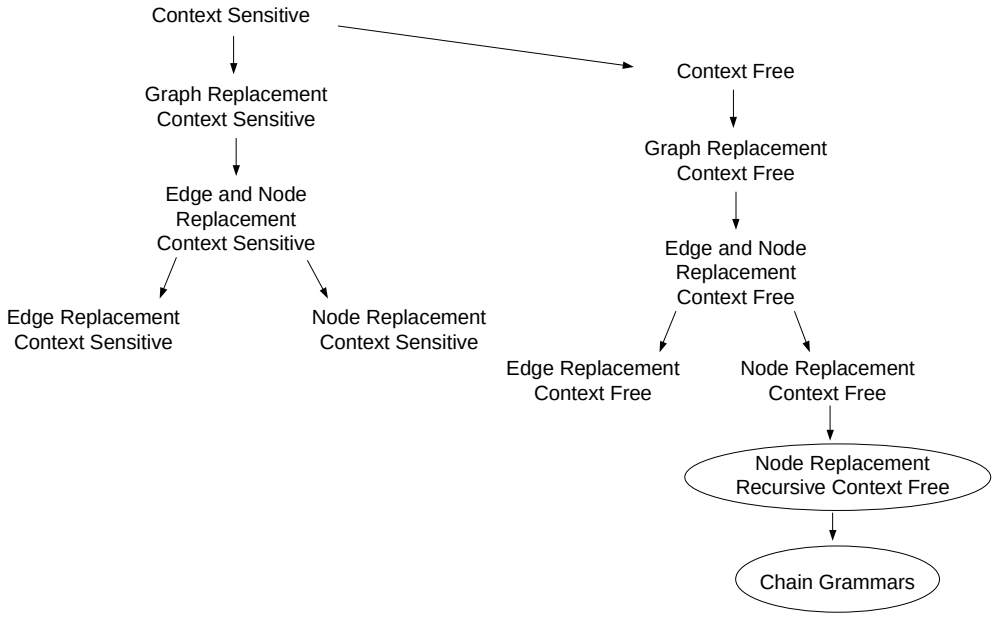


Figure 2: Hierarchy of graph grammars.

The grammar from Figure 3 consists of a graph isomorphic to three overlapping substructures and connection instructions. We find connection instructions when we check for overlaps. In this example there are two connection instructions, 1-3 and 1-4. Hence, in generation of a graph from the grammar, in every derivation step an isomorphic copy of the subgraph definition will be connected to the existing graph by connecting node 1 of the subgraph to either a node 3 or a node 4 in the existing graph. The grammar shown on the right in Figure 3 cannot only regenerate the graph shown on the left, but also generate generalizations of this graph. Generalization in this example means that the grammar describes graphs composed from one or more star looking substructures of four nodes labeled “a” and “b”. All these substructures overlap on a node with the label “a”.

In our algorithm we allow instances to overlap by one node only. Certainly, real data can contain instances which overlap on more than one node. We encountered this case in experiments with chemical structures. We also did studies where we allow instances to overlap by two nodes, which led us to consider inference of edge replacement recursive graph grammars. Allowing for larger overlap is an open research issue in inference of new classes of graph grammars which we have not yet explored. In this

paper we limit overlap to only one node. In the algorithm, we do not allow instances to grow further than overlap by one node.

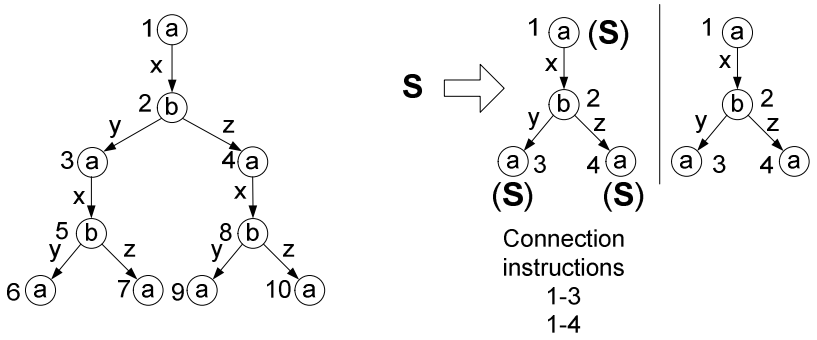


Figure 3: A graph with overlapping substructures and a graph grammar representation of it.

Algorithm 1 is our grammar discovery pseudocode. The function INFER_GRAMMAR is similar to the descriptions of Cook et al. [5] for substructure discovery and Jonyer et al. [10] for discovering grammars describing chains of isomorphic subgraphs connected by one edge. The input to the algorithm is a graph *G* which can be one connected graph or set of disconnected graphs. *G* can have directed edges or undirected edges. The algorithm assumes labels on nodes and edges. The algorithm processes the list of substructures *Q*. In Figure 4 we see an example of a substructure definition. A substructure consists of a graph definition and a set of instances from the input graph that are isomorphic to the graph definition. The example in Figure 4 is a continuation of the example in Figure 3. The numbers in parentheses refer to nodes of the graph in Figure 3.

The algorithm starts (line 3) with a list of substructures where every substructure is a single node and its instances are all nodes in the graph with this node label. The best substructure is initially the first substructure in the *Q* list (line 4). In line 8 we extend each substructure in *Q* in all possible ways by a single edge and a node or only by single edge if both nodes are already in the graph definition of the substructure. We allow instances to grow and overlap, but any two instances can overlap by only one node. We keep all extended substructures in *newQ*. We evaluate substructures in *newQ* in line 12. The recursive substructure *recursiveSub* is evaluated along with non-recursive substructures and is competing with non-recursive substructures. The total number of substructures considered is determined by the input parameter *Limit*. In line 19 we compress *G* with *bestSub.* Compression replaces every instance of *bestSub* with a single node. This node is labeled with a non-terminal label. The compressed graph is further processed until it cannot be compressed any more. In consecutive iterations *bestSub* can have one or more non-terminal labels. It allows us to create a hierarchy of grammar productions. The input parameter *Beam* specifies the width of a beam search, i.e., the length of Q. For more details about the algorithm see [5][10][11].

The function RECURSIFY_SUBSTRUCTURE takes substructure *S* and, if instances of *S* overlap, proposes recursive substructure *recursiveSub.* The list of connection instructions and the list of recursive instances are two main components of *recursiveSub*. We initialize them in line 1 and 2. We check for overlap in line 4. Figure 4 assists us in explaining conversion of substructure *S* into recursive substructure. Every instance graph has two positive integers assigned to it. One integer, in parentheses in Figure 4, is the number of a node in the processed graph *G*. The second integer is a node number of an instance graph. The instances are isomorphic to the substructure graph definition and instance node numbers are assigned to them according to this isomorphism. We check for overlap in line 4. Given pair of instances (I_1, I_2) we examine if there is a node $v \in G$, which also belongs to I_1 and I_2. We find two overlapping nodes, [3] and [4], examining node numbers in parentheses in the example in Figure 4. Having the number of node $v \in G$ we find corresponding to v two node numbers of instance graphs

Algorithm 1 Graph grammar discovery.

```
INFER_GRAMMAR (graph G, integer Beam, integer Limit)
1.  grammar={}
2.  repeat
3.     queue Q ={v | v is a node in G having a unique label}
4.     bestSub= first substructure in Q
5.     repeat
6.        newQ ={}
7.        for each substructure S ∈ Q
8.           newSubs = extend substructure S in all possible ways by a single edge and a node
9.           recursiveSub = RECURSIFY_SUBSTRUCTURE(S)
10.          newQ = newQ ∪ newSubs ∪ recursiveSub
11.          Limit=Limit-1
12.          evaluate substructures in newQ , maintain length(newQ) < Beam eliminating
       substructure with the lowest value if necessary
13.       end for
14.       if best substructure in newQ better than bestSub
15.       then bestSub = best substructure in newQ
16.       Q=newQ
17.    until Q is empty or Limit ≤ 0
18.    grammar = grammar ∪ bestSub
19.    G = G compressed by bestSub
20. until bestSub cannot compress the graph G
21. return grammar

RECURSIFY_SUBSTRUCTURE (substructure S)
1.  recursiveSub → connectionInstructionList = {}
2.  recursiveSub →Instances = {}
3.  for all pairs of instances (I1, I2),  I1∈ S, I2∈ S
4.     if (I1 and I2 overlap on node v∈ G )
5.        v_I = GET_INSTANCE_NODE(v, I1 )
6.        v'_I = GET_INSTANCE_NODE(v, I2 )
7.        if ( (v_I, v'_I) ∉ (recursiveSub → connectionInstructionList) )
8.           Add (v_I, v'_I) to (recursiveSub → connectionInstructionList)
9.        end if
10.       if I1 ∩ IRk ≠ Ø or I2 ∩ IRk ≠ Ø , where IRk is any member of recursiveSub →Instances
11.          modify IRk ,  IRk= IRk ∪ I1 ∪ I2 else
12.          create new entry IRk= I1 ∪ I2 and add it to recursiveSub →Instances
13.       end if
14.    end if
15. end for
16. return recursiveSub
```

$v_I \in I_1$ and $v'_I \in I_2$ (line 5 and 6). The pair of integers (v_I, v'_I) is a connection instruction. There are two connection instructions in Figure 4, 1-3 and 1-4. If (v_I, v'_I) is not already in the list of connections instructions for recursive substructure, we include it in line 8. The order in which we test instances in the

instance list for overlap does not matter. Even if one instance overlaps with two or more different instances on different nodes, overlap does not depend on the order of instances. Therefore, the connection instructions found from each overlap will be the same, no matter in which order we traverse the instance list.

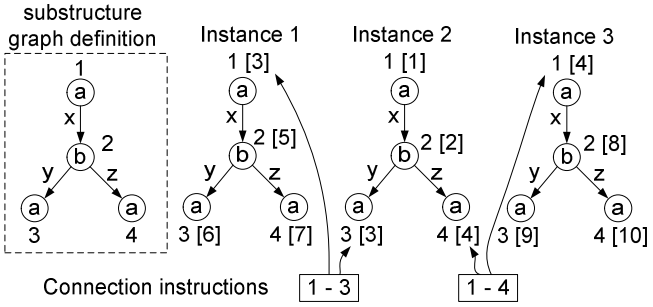


Figure 4: Substructure and its instances while determining connection instructions (continuation of the example from Figure 3).

We create the recursive substructure's instance list in lines 10 to 13 of RECURSIFY_SUBSTRUCTURE. A recursive instance is a connected subgraph of *G* which can be described by the discovered grammar production. It means that for every subset of instances $\{I_m, I_{m+1}, \ldots, I_l\}$ from the instance list of *S*, in which union $I_m \cup I_{m+1} \cup \ldots \cup I_l$ is a connected graph, we create one recursive instance $IR_k = I_m \cup I_{m+1} \cup \ldots \cup I_l$. The recursive instances are no longer isomorphic as instances of *S* and they vary in size. Recursive substructures compete with non-recursive. All instances described by a single recursive production are collectively replaced by a single node in the evaluation process; whereas, for non-recursive productions, each instance is replaced by a single node, one per instance. This method of tends to prefer recursive productions.

Subdue uses a heuristic search whose complexity is polynomial in the size of the input graph [5]. Our modification does not change the complexity of this algorithm. The overlap test is the main computationally expensive addition of our grammar discovery algorithm. Analyzing informally, the number of nodes of an instance graph is not larger than V, where V is the number of nodes in the input graph. Checking two instances for overlap will not take more than $O(V^2)$ time. The number of pairs of instances is no more than V^2, so the entire overlap test will not take more than $O(V^4)$ time.

6. Hierarchy of productions

In our first example from Figure 3, we described a grammar with only one production. Now we would like to introduce a complex example to illustrate the inference of a grammar which describes a more general tree structure. In Figure 5 we have a tree with all nodes having the same label. There are two repetitive subgraphs in the tree. One has three edges labeled "a," "b," and "c." The other has two edges with labels "x" and "y." There are also three edges K1, K2, and K3 which are not part of any repetitive subgraph. In the first iteration we find grammar production S1, because overlapping subgraphs with edges "a," "b," and "c" score the highest in compressing the graph. Examining production S1, we notice that node 3 is not involved in connection instructions. It is consistent with the input graph where there are no two subgraphs overlapping on this node. The compressed graph, at this point, contains the node S1, edges K1, K2, K3 and subgraphs with edges "x" and "y." In the second iteration our program finds all overlapping substructures with edges "x" and "y" and proposes production S2. Compressing the tree with production S2 results in a graph which we use as an initial production S, because the graph can be compressed no further. In Figure 5 productions for S1 and S2 have graphs as terminals. We will omit

drawing terminal graphs in subsequent figures. The tree used in this example was used in our experiments, and the grammar on the right in Figure 5 is the actual inferred grammar.

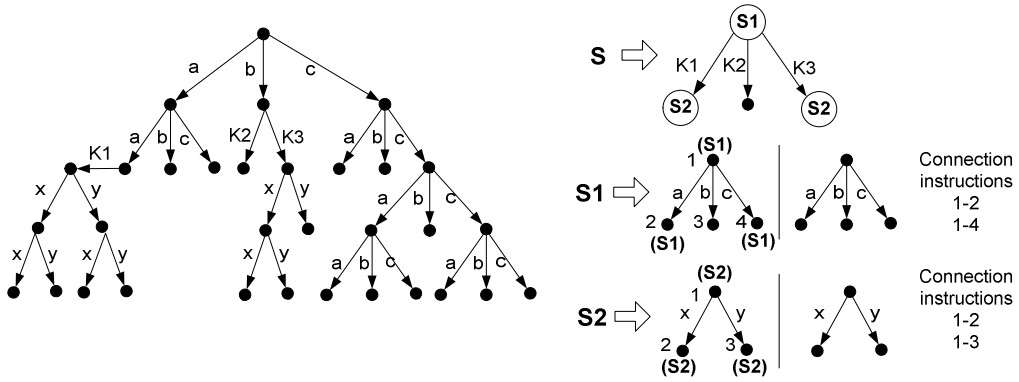


Figure 5: The tree (left) and inferred tree grammar (right).

7. Experiments

7.1 Methodology

Having our system implemented, we faced the challenge of evaluating its performance. There are an infinite number of grammars as well as graphs generated from these grammars. In our experiments we restricted grammars to node replacement grammars with two productions. The second production replaces a non-terminal node with a single terminal node. In Figure 6 we give an example of such a grammar. The grammar on the left is of the form used in generation. The grammar on the right is the inferred grammar in our experiment. The inferred grammar production is assumed to have a terminating alternative with the same structure as the recursive alternative, but with no non-terminals. We omit terminating production in Figure 6. We associate probabilities with productions used in generation. These probabilities define how often a particular production is used in derivations. Assigning probabilities to productions helps us to control the size of the generated graph. Our inference system does not infer probabilities. Oates et al. [17] addresses the problem of inferring probabilities assuming that the productions of a grammar are given. We are considering inferring probabilities along with productions as a future work.

We developed a graph generator to generate graphs from a known grammar. We can generate directed or undirected graphs with labels on nodes and edges. Our generator produces a graph by replacing a non-terminal node of a graph until all nodes and edges are terminal. The generation process expands the graph as long as there are any non-terminal edges or nodes. Since selection of a production is random according to the probability distribution specified in the input file, the number of nodes of a generated graph is also random. We place limits on the size of the generated graph with two parameters: minNodes and maxNodes. We generate graphs from the grammar until the number of nodes is between minNodes and maxNodes. We distinguish two different distortion operations to the graph generated from the grammar: corruption and added noise. Corruption involves the redirection of randomly selected edges. The number of edges of a graph multiplied by noise gives the number of redirected edges, where noise is a value from 0 to 1. We redirect an edge $e = (v_1, v_2)$ by replacing nodes v_1 and v_2 with two new, randomly selected graph nodes v'_1 and v'_2 .When we add noise, we do not destroy generated graph structure. We add new nodes and new edges with labels assigned randomly from labels used in already generated graph structure. We compute the number of added nodes from the formula (noise/(1- noise))*number_of_nodes. The number of added edges we find from (noise/(1- noise))*number_of_edges. A new edge connects two nodes selected randomly from existing nodes of the generated structure and newly added nodes. The distortion of a graph when we add noise is from added edges and vertices. The higher the noise value, the higher the probability that newly added nodes will

be connected to the rest of the graph. If the noise value is low, the new added nodes might not get connected to the graph and the distortion to the graph will be only from added edges.

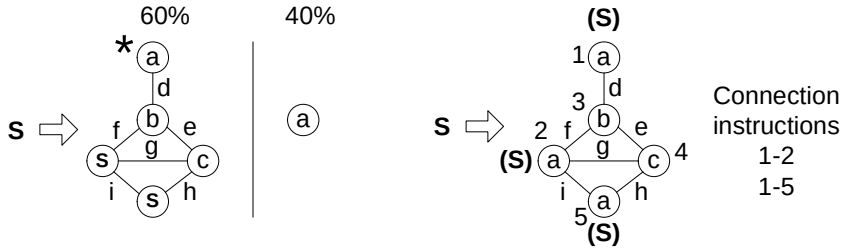


Figure 6: An example of a graph grammar used in the experiments.

We examined grammars with one, two, and three non-terminals. The first productions of the grammars have an undirected, connected graph with labels on nodes and edges on the right side. We use all possible connected simple graphs with three, four, and five nodes as the structures of graphs used in the productions. There are twenty nine different simple connected undirected unlabeled graphs [18]. We show them in Figure 9. Performing experiments with larger graph structures requires more computation time. Experiments with complex grammars which involve several productions we plan to describe in a separate report. We attempted several experiments with large social networks of generated terrorist networks; however, we did not find meaningful recursive patterns in this domain. For these experiments the input file had 8 graphs with a total number of vertices of 23,213 and a total number of edges of 34,132. Computations for this graph took 885 seconds. Our graph generator generates graphs from the known grammar that is based on one of the twenty-nine graph structures. Then we use our inference system to infer the grammar from the generated graph. We measure an error between the original and inferred grammar. We use MDL as a measure of the complexity of a grammar. Our results describe the dependency of the grammar inference error on complexity, noise, number of labels, and size of generated graphs.

7.2 MDL as a measure of complexity of a grammar

We seek to understand the relationship between graph grammar inference and grammar complexity, and so need a measure of grammar complexity. One such measure is the Minimum Description Length (MDL) of a graph, which is the minimum number of bits necessary to completely describe the graph. Here we define the MDL measure, which while not provably minimal, is designed to be a near-minimal encoding of a graph. See [5] for a more detailed discussion.

$\mathrm{MDL}(graph) = vbits + rbits + ebits$, where

vbits is the number of bits needed to encode the nodes and node labels of the graph

$vbits = \lg v + v \lg l_u$,

v is the number of nodes in the graph

$\lg v$ is the number of bits to encode the number of nodes v in the graph

l_u is the number of unique labels in the graph

rbits is the number of bits needed to encode the rows of the adjacency matrix of the graph

$$rbits = (v+1)\lg(b+1) + \sum_{i=1}^{v} \lg \binom{v}{k_i}$$

b is the maximum number of 1s in any row of the adjacency matrix

k_i is the number of 1s in a row i of the adjacency matrix

ebits is the number of bits needed to encode edges given in adjacency matrix

$ebits = e(1+\lg l_u) + (K+1)\lg m$,

e is the number of edges of a graph

m is the maximum number of edges between any two nodes; in our graphs m=1 because graphs are simple, therefore $(K+1)\lg m = 0$

K is number of 1s in adjacency matrix of a graph, in our graphs $K = e$

Since all the grammars in our experiments have two productions and the second production replaces a non-terminal with a single node, the complexity of the grammar will vary depending only on the graph on the right side of the first production. We would like our results for one, two and three non-terminal grammars to be comparable; therefore we do not want our measure of complexity of a grammar to be dependent on the number of non-terminals. In every graph used in the productions we reserve three nodes. We give the same label to these nodes. When we generate a graph, we replace one, two, or three labels of these nodes with the non-terminal S when we need a grammar with one, two or three non-terminals. However, when we measure MDL of a graph we leave the original three labels unchanged. In our experiments we always use that same non-terminal label. In the general case a production can contain different non-terminals. Every non-terminal would need to be counted as a different label of a graph and MDL would increase with increasing number of non-terminals. Therefore, replacing all non-terminals with one label eliminates the influence of the number of non-terminals on.the MDL measure, and we can compare inference error of the structures of the graphs used in productions across one, two and three non-terminals.

Next, we give an example of calculating the MDL of a graph using the graph structure from Figure 6. The adjacency matrix of the graph and the MDL calculation are as follows.

	a	*a*	*b*	*c*	*a*
a			1		
a			1	1	1
b				1	
c					1
a					

$vbits = \lg v + v\lg l_u = \lg 5 + 5\lg 9 = 18.17$

$rbits = (v+1)\lg(b+1) + \sum_{i=1}^{v} \lg\binom{v}{k_i} = (5+1)\lg(3+1) + \lg\binom{5}{1} + \lg\binom{5}{3} + \lg\binom{5}{1} + \lg\binom{5}{1} + \lg\binom{5}{0} = 22.29$

$ebits = e(1+\lg l_u) + (K+1)\lg m = 6(1+\lg 9) + (6+1)\lg 1 = 25.02$

$\mathrm{MDL}(graph) = vbits + rbits + ebits = 65.48$

We can compare this result with an MDL value 26.09 of a triangle with three vertices, three edges and four different labels.

7.3 Error

We introduce a measure to compare the original grammar to the inferred grammar. Our definition of an error has two aspects. First, there is the structural difference between the inferred and the original graph used in the productions. Second, there is the difference between the number of non-terminals and the number of connection instructions. If there is no error, the number of non-terminals in the original grammar is the same as the number of connection instructions in the inferred grammar. We compute the structural difference between graphs with an algorithm for inexact graph match initially proposed by Bunke and Allermann [2]. For more details see also [5]. We would like our error to be a value between 0

and 1; therefore, we normalize the error by having in the denominator the sum of the size of the graph used in the original grammar and the number of non-terminals. We do not allow an error to be larger than 1; therefore, we take the minimum of 1 and our measure as a final value. The restriction that the error is not larger than 1 prohibits unnecessary influence on the average error taken from several values by inferred graph structure significantly larger than the graph used in the original grammar.

$$Error = \min\left(1, \frac{\text{matchCost}(g_1, g_2) + |\#CI - \#NT|}{\text{size}(g_1) + \#NT}\right), \qquad \text{where}$$

$\text{matchCost}(g_1, g_2)$ is the minimal number of operations required to transform g_1 to a graph isomorphic to g_2, or g_2 to a graph isomorphic to g_1. The operations are: insertion of an edge or node, deletion of a node or an edge, or substitution of a node or edge label.

$\#CI$ is the number of inferred connection instructions

$\#NT$ is the number of non-terminals in the original grammar

$\text{size}(g_1)$ is the sum of the number of nodes and edges in the graph used in the grammar production

Most of the time $\text{matchCost}(g_1, g_2)$ is only a fraction of the size of the original grammar. We could choose a larger denominator to ensure that error will not be larger than 1. However, in some cases the learned grammar was an attached pair of the structure in the original grammar, as in Figure 10, where the language of the learned grammar is very close to the original grammar, but the structural difference between the learned and original grammars is large. We decided on $\text{size}(g_1) + \#NT$ and limit error to 1, because it gave a clearer picture of the error. Using $\text{size}(g_1) + size(g_2) + \#NT$ as a denominator would also be a valid measure, and be between 0 and 1, but may unduly penalize for variants of the target grammar..

7.4 Experiment 1: Error as a function of noise and complexity of a grammar

We used twenty nine graphs from Figure 9 in grammar productions. We assigned different labels to nodes and edges of these graphs except three nodes used for non-terminals. We generated graphs with noise from 0 to 0.9 in 0.1 increments. For every value of noise and MDL we generated thirty graphs from the known grammar and inferred the grammar from the generated graph. We computed the inference error and averaged it over thirty examples. We generated 8700 graphs to plot each of the three graphs in Figure 7**.** The first plot shows results for grammars with one non-terminal. The second and the third plot show results for grammars with two and three non-terminals. We did not corrupt the generated graph structure in experiments in Figure 7. As noise we added nodes and edges to the generated graph structure. We used only the ADD_NOISE_TO_GRAPH function of our generator. Figure 8 has the same results as Figure 7 with the difference that we corrupted the graph structure generated from the grammar and then we added nodes and edges to the graph. We used both CORRUPT_GRAPH_STRUCTURE and ADD_NOISE_TO_GRAPH functions of the generator to distort the graph.

We see trends in the plots in Figure 7 and Figure 8 Error decreases as MDL increases. A low value of MDL is associated with small graphs, with three or four nodes and a few edges. These graphs, when used on the right hand side of a grammar production, generate graphs with fewer labels than grammars with high MDL. Smaller numbers of labels in the graph increase the inference error, because everything in the graph looks similar, and the approach is more likely to propose another grammar which is very different than the original. As expected, the error increases as the noise increases in experiments with corrupted graph structure. However, there is little dependency of an error from the noise if the graph generated from the grammar is not corrupted (Figure 7).

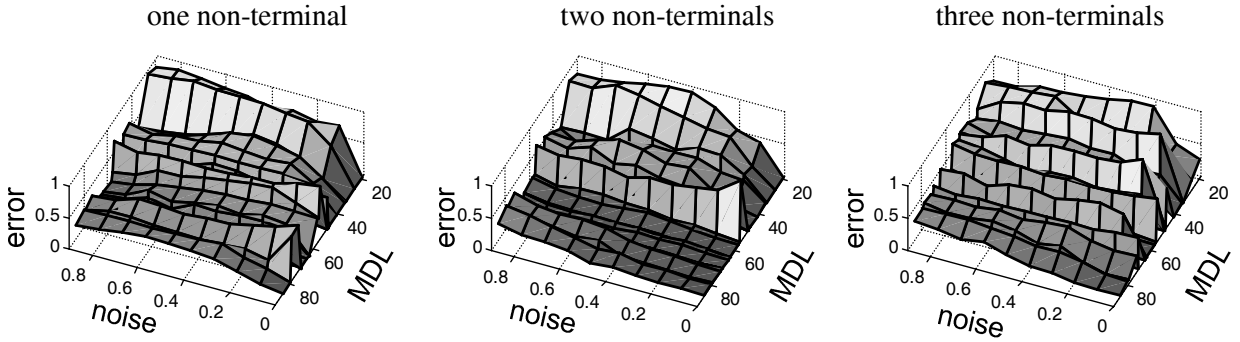


Figure 7: Error as a function of noise and MDL where graph structure was not corrupted.

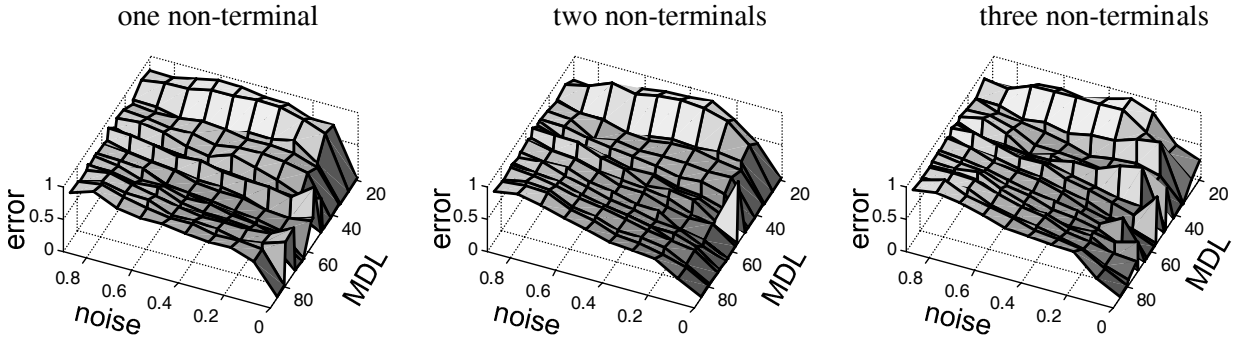


Figure 8: Error as a function of noise and MDL where graph structure was corrupted.

We average the value of an error over ten values of noise which gives us the value we can associate with the graph structure. It allowed us to order graph structures used in the grammar productions based on average inference error. In Figure 9 we show all twenty nine connected simple graphs with three, four and five nodes used in productions ordered in non-decreasing MDL value of a graph structure. We decided on this order because graphs we produce show dependency of error on MDL of a graph structure. This order gives us the sense of complexity. In Table 2 we give an order of graph structures for six experiments with corrupted and non-corrupted structures and one, two, and three non-terminals. The numbers in the table refer to structure numbers in Figure 9. We see in Table 2 that graph number 21 is close to the beginning of the list in all six experiments. Graphs number 1, 2, and 11 are close to the end of all six lists. We conclude that when graph number 21 is used in the grammar production, it is the easiest for our inference algorithm to find the correct grammar. When graph number 1, 2, or 11 is used in the grammar production and generated graphs have noise present, we infer grammars with some error. We also observe a tendency of decreasing error with increasing MDL in the graph orders in Table 2. Graph 29 has the highest MDL, because it has the most nodes and edges. In five experiments graph 29 is closer to the end of the list.

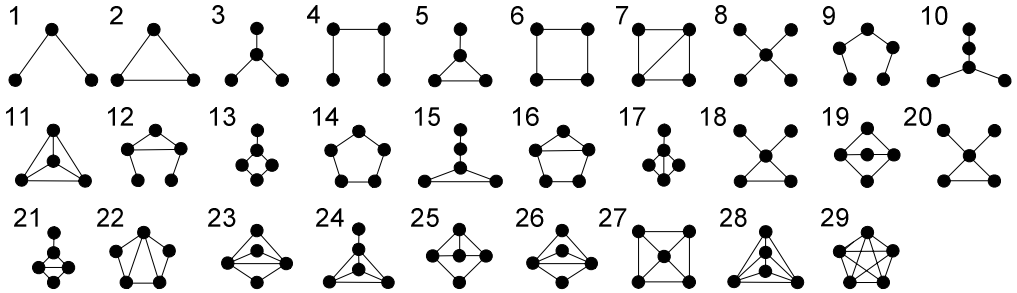


Figure 9: Twenty nine simple connected graphs ordered according to non-decreasing MDL value.

Table 2: Twenty nine simple graphs ordered according to increasing average inference error of six experiments in Figure 7 and Figure 8. The numbers in the table refer to structures in Figure 9.

Number of non-terminals			
1	Not Corrupted	21 17 22 15 8 10 23 28 20 27 29 19 26 12 16 3 18 4 24 25 9 5 7 14 6 13 11 1 2
2	Not Corrupted	21 23 22 15 18 16 17 20 19 9 28 12 10 14 26 13 27 25 8 24 29 4 5 7 3 6 11 2 1
3	Not Corrupted	21 15 23 16 17 19 18 14 9 13 28 12 27 26 25 24 5 10 4 29 22 6 20 7 11 2 8 1 3
1	Corrupted	8 10 12 21 17 15 20 23 16 19 18 22 13 14 9 27 4 28 25 3 7 29 24 6 26 5 11 1 2
2	Corrupted	9 17 19 16 21 13 18 8 15 14 10 12 25 27 23 22 24 20 26 28 4 3 6 5 29 7 11 1 2
3	Corrupted	9 19 14 12 18 16 13 15 21 17 4 23 10 25 27 26 5 6 24 20 28 22 29 8 7 3 11 1 2

Quantitative definition of an error allows us to automate the process and perform tests on thousands of graphs. The error is caused by a wrongly inferred graph structure used in the production or number of connection instructions which is too large or too small. However, there are cases where the inferred grammar represents the graph well, but the graph in the production has a different structure. For example, we observed that the grammar with MDL=55.58 and graph number 11 causes an error even if we infer the grammar from graphs with no corruption and zero noise. The inferred graph structure contains two overlapping copies of the graph used in the original grammar production. We illustrate it in Figure 10. The structure has significant error, yet does subjectively capture the recursive structure of the original grammar.

7.5 Experiment 2: Error as a function of number of labels and complexity of a grammar

We would like to evaluate how error depends on the number of different labels used in a grammar. We restricted graph structures used in productions to graphs with five nodes. Every graph structure we labeled with 1, 2, 3, 4, 5 or 6 different labels. For every value of MDL and number of labels we generated 30 different graphs from the grammar and computed average error between them and the learned grammars. The generated graphs were without corruption and without noise. We show the results for one, two, and three non-terminals in Figure 13. Below the three dimensional plots, for clarity, we give two dimensional plots with triangles representing the errors. The larger and lighter the triangle the larger is the error. We see that the error increases as the number of different labels decreases. We see on the two dimensional plots the shift in error towards graphs with higher MDL when the number of non-terminals increases.

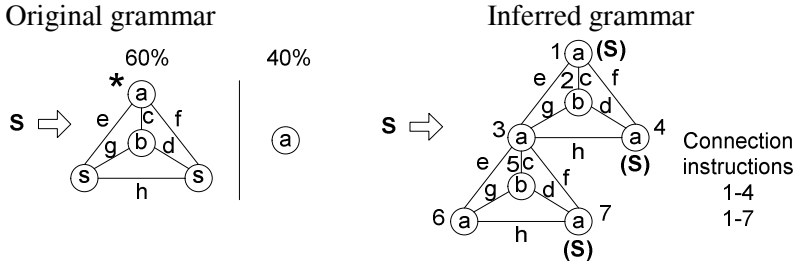


Figure 10: An inference error where larger graph structure is proposed.

The average error for graphs with only one label is 1 or very close to 1. The most frequent inference error results from the tendency of our algorithm to propose one-edge grammars when inferred from a

graph with only one label. We illustrate this in Figure 11 where we see a production with a pentagon using only one label "a". The inferred grammar has one edge with two connection instructions 1-1 and 1-2. Since all the edges in the generated graph have the same label and all the nodes have the same label, this grammar compresses the graph well and is evaluated highly by our compression-based measure. However, this one-edge grammar cannot generate even a single pentagon. An evaluation measure which penalizes grammars for an inability to generate an input graph would bias the algorithm away from single-edge grammars and could correct the one-edge grammar problem. However, this approach would require graph-grammar parsing, which is computationally complex

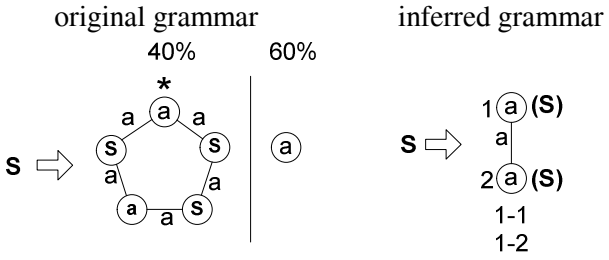


Figure 11: Error where inferred grammar is reduced to production with single edge.

7.6 Experiment 3: Error as a function of size of a graph and complexity of a grammar

We generated graphs from grammars with two non-terminals and noise=0.2. The number of nodes of the generated graphs was from the interval [x, x+20], where we change x from 20 to 420. For each value of x and MDL we generated thirty graphs and compute average inference error over them. We show in Figure 14 the results for corrupted and not corrupted graph structure. We concluded that there is no dependency between the size of a sample graph and inference error.

7.7 Experiment 4: Limitations

In Figure 12 we show an example illustrating the limits of our approach. In Figure 12 (a) we have a graph consisting of overlapping squares. All labels on nodes are the same, and we omit them. The squares do not overlap by one node but by an edge. Our algorithm assumes that only one node overlaps in the instances of the substructure and therefore infers the grammar shown in Figure 12 (b). The inferred grammar can generate chains, an example of which is shown in Figure 4 (c). The original input graph is not in the set of graphs generated by the inferred grammar. An extension of our method to overlapping edges would allow us to infer the correct grammar in this example.

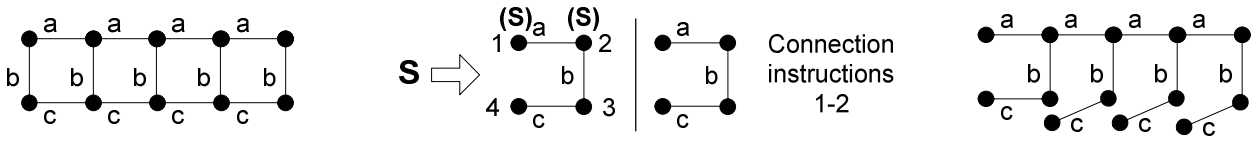


Figure 12: Graph with overlapping squares (a), inferred grammar (b), and graph generated from inferred grammar (c)

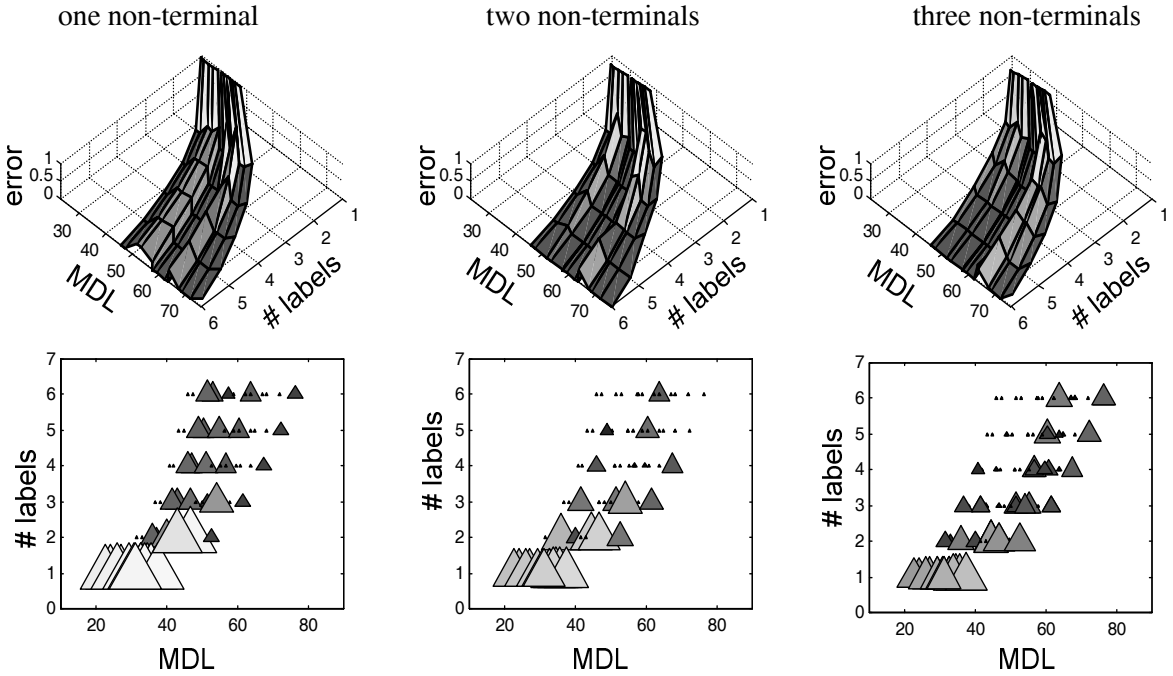
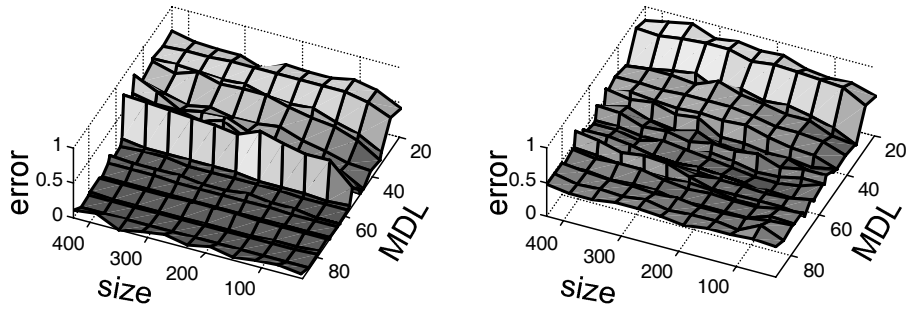


Figure 13 : Error as a function of MDL and number of different labels used in a grammar definition.



(a) (b)

Figure 14: Error as a function of MDL and size of generated graphs (noise=0.2, two non-terminals): (a) uncorrupted graph structure, (b) corrupted graph structure

Figure 15 shows another example illustrating the limits of our algorithm. The first graph in the first production on the left is a square with two non-terminals labeled S1, and the graph of the second production is a triangle with one non-terminal labeled S. Our algorithm is not designed to find alternating productions of this type. We generated a graph from the grammar on the left, and the grammar we inferred is on the right in Figure 15. The inferred grammar has one production in which the graph combines both the triangle and square. The set of graphs generated by alternating squares and triangles according to the grammar from the left does not match exactly the set of graphs of the inferred grammar. Nevertheless, we consider it an accurate inference, because the inferred grammar will describe the majority of every graph generated by the original grammar. If we were learning alternating productions, we would need to infer them in separate iterations and analyze connection instructions, not in every iteration as we do now, but once for all interactions combined.

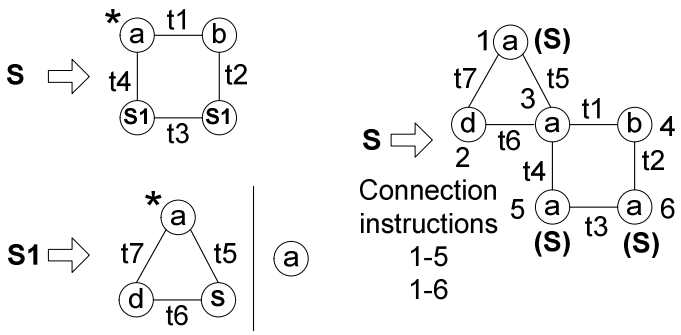


Figure 15: The grammar with alternating productions (left) and inferred grammar (right).

7.8 Experiment 5: Chemical structures

As an example from the real-world domain of chemistry, we use four chemical structures as the input graphs in our next experiment. Figure 16 and Figure 17 show the structures of the molecules and the grammar productions we found in these structures. The first structure in Figure 16 is the structure of cellulose with hydrogen bonding. The second molecule is macrocyclic gallium carboxylate [23]. We found a grammar production with the Ga-Ga bond. The graph used in the production definition appears four times in the structure. The third structure in Figure 16 is water-soluble tin-based metallodendrimer [21]. We inferred two productions. We found production S1 in the first iteration. Production S1 has connection instruction 1-1 which means that vertex number 1 is replaced by an isomorphic instance of the right hand side of the S1 production, and the connecting vertex in the new instance of a graph is also vertex 1. We found the second production after all instances of S1 were compressed into single vertices. The graph on the right hand side of production S is a graph of a chemical structure compressed with S1.

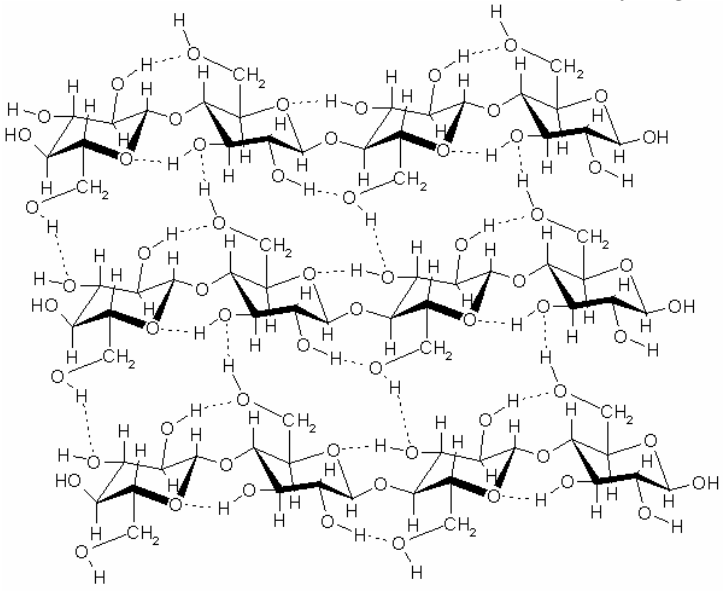
In Figure 17 we have the structure of a dendronized polymer [25]. Its graph grammar representation consists of three productions. Zhang et al. describe several chemical structures where the graph we found in production S2 in Figure 17 appears two, six, and fourteen times. Since production S1 conveys the idea of "one or more" connected graphs of the S1 structure, it intends to describe the entire family of chemical structures described in Zhang et al.'s paper. The grammar productions we found capture the underlying motifs of the chemical structures. They show the repetitive connected components, the basic building blocks of the structures. We can search for such underlying building block motifs in different domains, hoping that they will improve our understanding of chemical, biological, computer, and social networks.

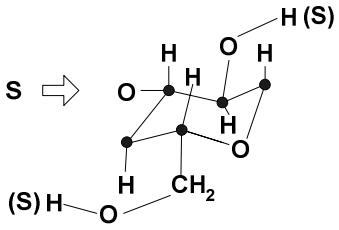
7.9 Experiment 6: Inferring XML schema

XML has become a common format for data exchange on the Internet. XML schema or DTDs define the structure and constraints on XML documents. Developers often need to write schema for existing XML documents, and so they would find a schema inference system practical. The structure of an XML file is an example of relational data with repetitive recursive patterns we can represent as graph grammars. We developed a converter which converts an XML file into a tree. We used the Java implementation of Document Object Model (DOM) in our converter. According to [1] there are twelve DOM node types: Element, Attr, Text, CDATASection, EntityReference, Entity, ProcessingInstruction, Comment, Document, DocumentType, DocumentFragment, and Notation. However in our implementation we build a directed tree with root node always labeled DOC and then the only type of data we extract in the examples below is 'Element'. From the perspective of our graph grammar inference system we need a pattern which is repeated in the graph, so we eliminated unique text data (e.g., names, card numbers, and price values). We do not assign labels to the edges of the tree.

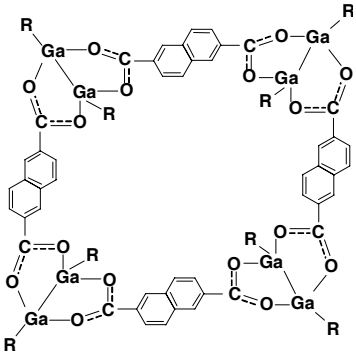
Chemical structure Inferred productions

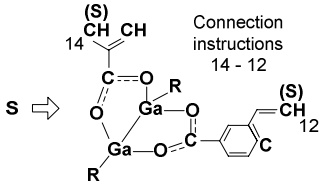
cellulose with hydrogen bonding



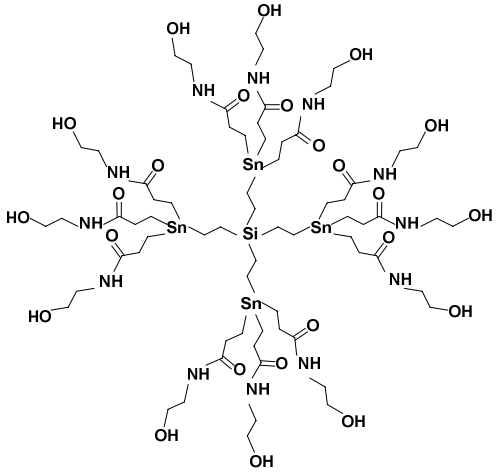


Macrocyclic gallium carboxylate





water-soluble tin-based metallodendrimer $Si[CH_2CH_2Sn(CH_2CH_2CONHCH_2CH_2OH)_3)]_4$



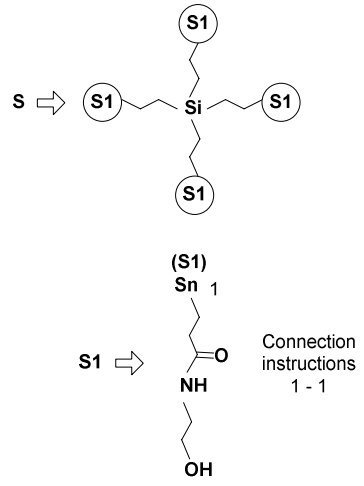


Figure 16: Three chemical structures (left) and the inferred grammar production (right).

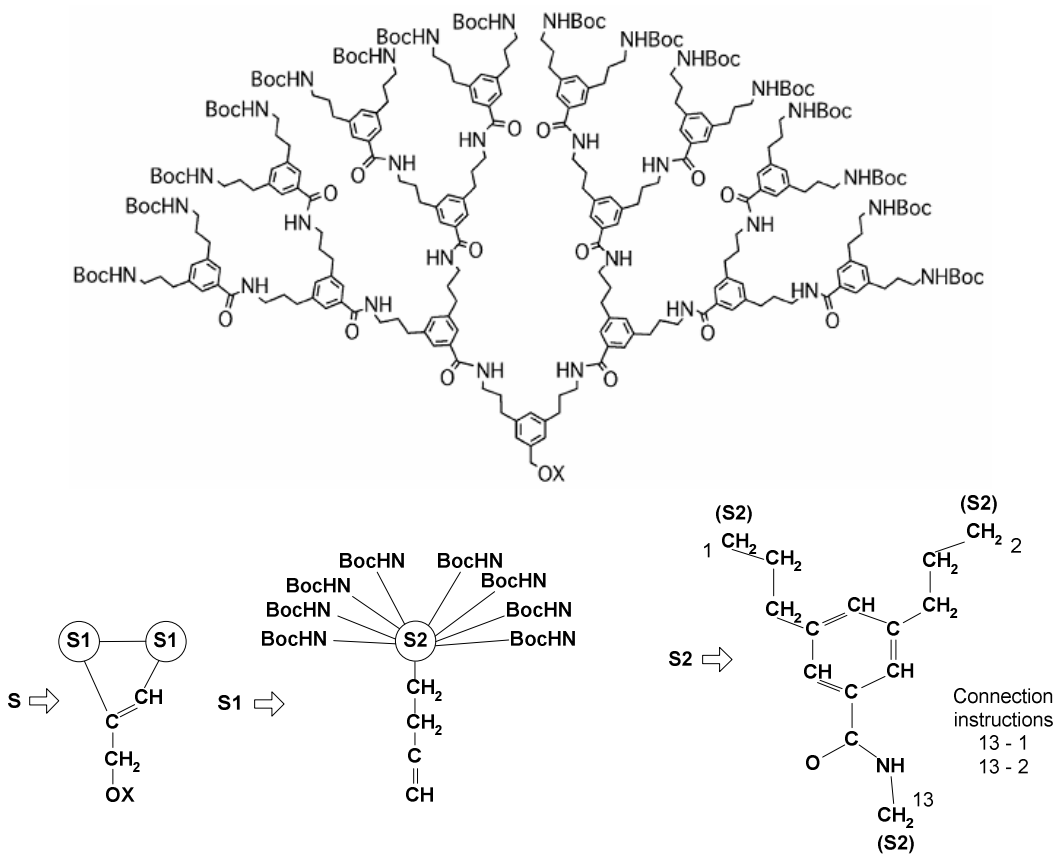


Figure 17: The structure of dendronized polymer and its representation in hierarchical graph grammar productions.

We selected domains for our experiments where we can easily identify the meaning of results. XML files often contain data with repetitive structures. An example of the beginning of such a file with pharmacology data and its Document Type Definition (DTD) is show in Figure 18. This is a sample file found on the National Library of Medicine website. The tree after conversion of this file we show in Figure 19. In Figure 20 we see the graph grammar found by our inference system. Productions S1 and S2 give the representation of a structure of an XML file which contains pharmacological data. Examining the DTD we see that element *PharmacologicalActionSubstanceSet* contains zero or more *Substance* elements. It is indicated by '*'. Similarly, element *PharmacologicalActionList* contains one or more *PharmacologicalActionOfSubstance* elements. It is marked in the DTD with a '+'. We discover these two concepts in production S1 and S2 in Figure 20. However, our inference system cannot distinguish between 'one or more' and 'zero or more' concepts. In the generation process we would connect the node labeled S2 of one instance of the production graph to the node with the same label of another instance. The process would continue until the node labeled S2 would be replaced by label *PharmacologicalActionSubstanceSet*. Production S1 is included as a non-terminal node of production S2. In our present implementation we do not specify to which node of a graph on the right hand side of S1 we would connect *Substance* node of S2. Following from parent to child nodes of structure of graphs of S1 and S2 we can find corresponding DTD entries. For example *PharmacologicalActionOfSubstance* has only one child *DescriptorReferredTo*, which corresponds to the DTD entry <!ELEMENT PharmacologicalActionOfSubstance (DescriptorReferredTo)>. We conclude that the recursive patters we found adequately represent the organization of the XML file in this domain. The connections instructions properly point out at nodes where recursion accrues. The inclusion of production S1 to S2 also properly shows hierarchy in the data.

```
<?xml version="1.0"?>
<!-- Sample for pa_substance2006.xml  -->
<!DOCTYPE
PharmacologicalActionSubstanceSet SYSTEM
"pa_substance2006.dtd">
<!-- Root element -->
<PharmacologicalActionSubstanceSet>
  <!-- Substance 1 (Descriptor) -->
  <Substance>
    <RecordUI>D000536</RecordUI>
    <RecordName>
      <String>Aluminum Hydroxide</String>
    </RecordName>
    <!-- The list of PAs for this substance -->
    <PharmacologicalActionList>
      <!-- First PA -->
      <PharmacologicalActionOfSubstance>
        <DescriptorReferredTo>

<DescriptorUI>D000276</DescriptorUI>
          <DescriptorName>
            <String>Adjuvants,
Immunologic</String>
          </DescriptorName>
        </DescriptorReferredTo>
      </PharmacologicalActionOfSubstance>
….
```

```
<!ENTITY  % DescriptorReference
"(DescriptorUI,
DescriptorName)">
<!ELEMENT
PharmacologicalActionSubstanceS
et (Substance*)>
<!ELEMENT Substance
((RecordUI,RecordName),
PharmacologicalActionList)+>
<!ELEMENT
PharmacologicalActionList
(PharmacologicalActionOfSubstan
ce)+>
<!ELEMENT
PharmacologicalActionOfSubstanc
e (DescriptorReferredTo)>

<!ELEMENT DescriptorReferredTo
(%DescriptorReference;)>
<!ELEMENT DescriptorUI
(#PCDATA)>
<!ELEMENT DescriptorName
(String)>
<!ELEMENT RecordUI (#PCDATA) >
<!ELEMENT RecordName (String) >
<!ELEMENT String (#PCDATA)>
```

Figure 18: An XML file describing pharmacology data and the Document Type Definition (DTD)

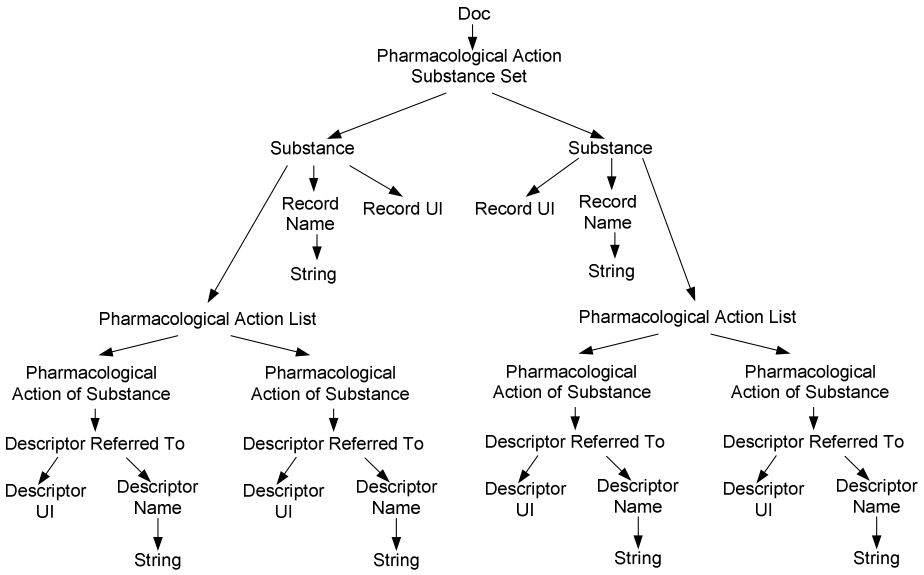


Figure 19: A graph representation of an XML file.

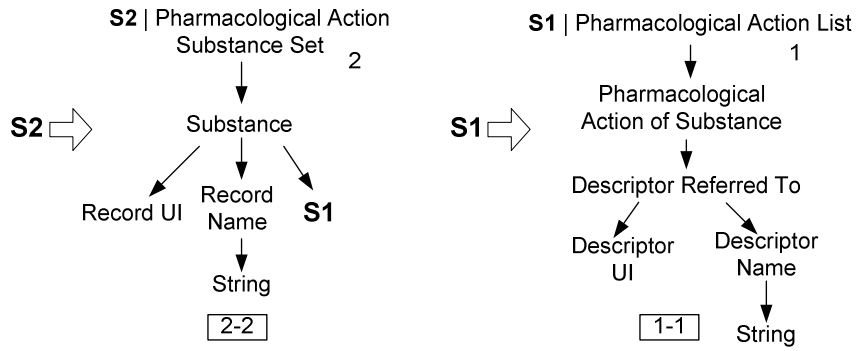


Figure 20: Graph grammar found by inference system from the XML tree.

8. Conclusions and future work

We described an algorithm for inferring certain types of graph grammars we call recursive node replacement graph grammars. The algorithm is based on previous work in frequent substructure discovery. It checks if frequent subgraphs overlap by a node and proposes a graph grammar if they do. The algorithm we described has its limitations: the left side of the production is limited to one single node; only connecting two single nodes is allowed in derivations. The algorithm finds recursive productions if repetitive patterns occur within an input graph and they overlap. If such patterns do not exist, the algorithm finds non-recursive productions and builds a hierarchical structure of the input data. Grammar productions with graphs of higher complexity measured by MDL are inferred with smaller error. There is little dependency of error on noise if the generated graphs are not corrupted. The error of grammar inference increases as the number of different labels used in the grammar decreases. There is no dependency between the size of a sample graph and inference error. If all labels on nodes are the same and all labels on edges are the same, the algorithm produces a grammar which has only one edge in the graph definition. One-edge grammars over-generalize if the input graph is a tree, and they are inaccurate in many other graphs. This tendency to find one-edge grammars from large, connected graphs is due to the fact that one-edge grammars score high because they can compress the entire graph.

In the pharmacy domain we found two recursive concepts: Pharmacological Action Substance Set containing one or more Substance elements, and Pharmacological Action List containing one or more Pharmacological Action Substance. We showed that the graph grammar inference algorithm can extract the organization and hierarchy of the structure of XML files. We compared the inferred graph grammar to the DTD noticing correspondence between DTD statements and graph grammar productions.

Grammars inferred by the approach developed by Jonyer et al. [10][11] were limited to chains of isomorphic subgraphs which must be connected by a single edge. Since the connecting edge can be included in the production's subgraph, and isomorphic subgraphs will overlap by one vertex, our approach can infer Jonyer et al.'s class of grammars. As we noticed in our experiment shown in Figure 12, when the subgraphs overlap by more than one node, our algorithm still looks for overlap on only one node and infers a grammar which cannot generate the input graph. Therefore one extension to the algorithm would be a modification which would allow for overlap larger than a single node.

The evaluation method can be modified to avoid one-edge grammar productions in graphs with one label. We are exploring other domains where data can be represented as a graph composed from smaller structures to further test our inference system and examine it as a data mining tool for these domains. We are continuing our research in graph grammar inference to develop methods allowing for discovery of more powerful classes of graph grammars than discussed in this paper.

References

[1] K. Ahmed., S. Ancha, A. Cioroianu, J. Cousins, J. Crosbie, J. Davies, K. Gabhart, S. Gould, R. Laddad, S. Li, B. Macmillan, D. Rivers-Moore, J. Skubal, K. Watson, S. Williams, and J. Hart, 2001, Professional Java XML, WROX.

[2] H. Bunke, G. Allermann, *Inexact graph matching for structural pattern recognition.* Pattern Recognition Letters, 1(4) 245-253. 1983

[3] R. Chittimoori, L. B. Holder, and D. J. Cook. *Applying the subdue substructure discovery system to the chemical toxicity domain*. In In the Proceedings 50 of the Twelfth International Florida AI Research Society Conference, 90-94, 1999.

[4] D. Cook and L. Holder, *Graph-Based Data Mining.* IEEE Intelligent Systems, 15(2), pages 32-41, 2000.

[5] D. Cook and L. Holder, *Substructure Discovery Using Minimum Description Length and Background Knowledge.* Journal of Artificial Intelligence Research, Vol 1, (1994), 231-255, 1994

[6] S. Doshi, F. Huang, and T. Oates, *Inferring the Structure of Graph Grammar from Data.* Proceedings of the International Conference on Knowledge Based Computer Systems (KBCS'02) 2002.

[7] D. Gernert, *Graph grammars as an analytical tool in physics and biology*. Biosystems 1997, vol. 43, no. 3, pp. 179-187(9), 1997

[8] J. A. Gonzalez, L. B. Holder, and D. J. Cook. *Structural knowledge discovery used to analyze earthquake activity*. In Proceedings of the Thirteenth Annual Florida AI Research Symposium, 2000.

[9] E. Jeltsch, H. Kreowski, *Grammatical Inference Based on Hyperedge Replacement. Graph-Grammars.* Lecture Notes in Computer Science 532, 1990: 461-474, 1990

[10] I. Jonyer, L. Holder, and D. Cook, *Concept Formation Using Graph Grammars*, Proceedings of the KDD Workshop on Multi-Relational Data Mining, 2002.

[11] I. Jonyer and L. Holder, and D. Cook, *MDL-Based Context-Free Graph Grammar Induction and Applications*. International Journal of Artificial Intelligence Tools, Volume 13, No. 1 pages 65-79, 2004.

[12] C. Kim, *A hierarchy of eNCE families of graph languages.* Theoretical Computer Science 186, 157-169, 1997.

[13] M. Kuramochi and G. Karypis, *Frequent subgraph discovery.* In Proceedings of IEEE 2001 International Conference on Data Mining (ICDM '01), 313-320, 2001.

[14] R. Mehta, D. J. Cook, and L. B. Holder. I*dentifying inhabitants of an intelligent environment using a graph-based data mining systems.* In Proceedings of the Florida Artifcial Intelligence Research Symposium, 2003.

[15] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon1, *Network Motifs: Simple Building Blocks of Complex Networks*, *Science*. Vol 298, Issue 5594, 824-827 , 2002

[16] G. Nevill-Manning and H. Witten*, Identifying Hierarchical Structure in Sequences: A linear-time algorithm.* Journal of Artificial Intelligence Research, Vol 7, (1997), 67-82, 1997

[17] T. Oates, S. Doshi, and F. Huang. *Estimating Maximum Likelihood Parameters for Stochastic Context-Free Graph Grammars.* In T. Horváth and A. Yamamoto, editors, Proceedings of the 13th International Conference on Inductive Logic Programming, volume 2835 of Lecture Notes in Artificial Intelligence, pages 281--298. Springer-Verlag, 2003.

[18] R Read and R. Wilson, *An Atlas of Graphs*. Oxford University Press, 1998

[19] J. Rissanen,. `*Modeling by shortest data description*, Automatica, Vol. 14, pp. 465-471

[20] Y. Sakakibara, *Recent advances of grammatical inference*. Theoretical Computer Science, 185:15-45, 1997.

[21] H Schumann, B Wassermann, S Schutte, J Velder, Y Aksu, W. Krause, and B Radüchel, *Synthesis and Characterization of Water-Soluble Tin-Based Metallodendrimers.* Organometallics, 22, 2034-2041, 2003

[22] S. Su, D. J. Cook, and L. B. Holder. *Knowledge discovery in molecular biology: Identifying structural regularities in proteins.* Intelligent Data Analysis, 3, 413-436, 1999.

[23] W Uhl, A Fick, Thomas Spies, Gertraud Geiseler, and Klaus Harms, Gallium-Gallium *Bonds as Key Building Blocks for the Formation of Large Organometallic Macrocycles, on the Way to a Mesoporous Molecule*. Organometallics, 23 (1), 72 -75, 2004

[24] X. Yan and J. Han, *gSpan: Graph-based substructure pattern mining.* In IEEE International Conference on Data Mining, Maebashi City, Japan, December 2002.

[25] A. Zhang, B. Zhang, E. Wachtersbach, M. Schmidt, and A. Schluter. *Efficient synthesis of high molar mass, first- fourth-generation distributed dendronized polymers by the macromonomer approach*. Chemistry a European Jouranl, 9(24), 6083-6092, 2003