International Journal on Artificial Intelligence Tools
Vol. 17, No. 3 (2008) 539–554
© World Scientific Publishing Company

INFERENCE OF EDGE REPLACEMENT GRAPH GRAMMARS

JACEK P. KUKLUK
Department of Radiation Oncology
Dana-Farber/Brigham and Women's Cancer Center
Harvard Medical School
75 Francis Street, Boston, MA 02115
jkukluk@lroc.harvard.edu

LAWRENCE B. HOLDER and DIANE J. COOK
School of Electrical Engineering and Computer Science
Washington State University
Box 642752, Pullman, WA 99164
holder@wsu.edu, cook@eecs.wsu.edu

We describe an algorithm and experiments for inference of edge replacement graph grammars. This method generates candidate recursive graph grammar productions based on isomorphic subgraphs which overlap by two nodes. If there is no edge between the two overlapping nodes, the method generates a recursive graph grammar production with a virtual edge. We guide the search for the graph grammar based on the size of the grammar and the portion of the graph described by the grammar. We show experiments where we generate graphs from known graph grammars, use our method to infer the grammar from the generated graphs, and then measure the error between the original and inferred grammars. Experiments show that the method performs well on several types of grammars, and specifically that error decreases with increased numbers of unique labels in the graph.

Keywords: Grammar induction; graph grammars; graph mining.

1. Introduction

There is overlap in the recurring patterns or motifs representing the building blocks of networks in nature. Palla *et al.*[14] point out the existence of an overlap between parts of graphs representing social networks and proteins. They call them overlapping communities. Most knowledge discovery and data mining approaches look for independent recurring patterns, but do not consider how these patterns can connect and overlap iteratively or recursively to generate arbitrary-sized relational data. Graph grammars provide a representation for such knowledge. 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.[4,11] Examples of applicable domains are: chemical structures, XML files, and biological networks.[9] In our method of graph grammar inference we search for overlap between isomorphic subgraphs of a graph. The overlap allows our method to propose recursive graph-grammar productions. The first approach was to search for overlap by a single node, which led to developing a system for inference of Node Replacement Recursive Graph Grammars.[9] In this paper we describe an approach that allows inference of Edge Replacement Recursive Graph Grammars. In the next section we will describe related work. Then, we define the class of grammars, describe the inference algorithm and discuss inference error. We also address how different numbers of labels used in the graph affect the inference error. Experiments with the chemical structure of G tetrad and conclusions close the paper.

2. Related Work

Jeltsch and Kreowski[5] analyzed theoretically the inference of hyperedge replacement graph grammars introducing operations on a set of undirected, unlabeled graphs which guarantee that the inferred grammar can generate the input set of graphs. Oates, Doshi, and Huang[16] assume that the structure of a graph of a hyperedge replacement context free graph grammar is given. They are interested in inference of probabilities associated with every rule of a grammar. Nevill-Manning and Witten[13] developed SEQUITUR which works on strings, but their approach is similar to ours in the sense that it infers hierarchical structure by replacing substrings by 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.

Jonyer, Holder, and Cook[6,7] developed an approach to infer node replacement graph grammars which describe graphs in the form of "chains", where isomorphic copies of subgraphs are adjacent to each other in a chain connected by a single edge. Their algorithm 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 descriptions 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 that can describe tree structures, while Jonyer's approach does not allow for tree grammars. We conducted experiments with Jonyer's approach, called SubdueGL, to illustrate the types of graph grammars it can find and its limitations. We generated graphs from the grammar and then used SubdueGL to infer this grammar. We show our results in Figure 1. In this figure, above the productions, we indicated a percentage that signifies the probability with which we are using every production. In Figure 1 we show a grammar that generates squares and triangles connected in series. Every square or triangle is connected to another square or triangle by one edge. The edge that connects the patterns is labeled nx. The labels on the vertices and edges of the patterns are distinct. 40% probability is assigned to both the nonterminal square and nonterminal triangle. Terminal square and triangle are assigned probability 10%. In Figure 1, below the grammar we drew one generated graph. It contains four squares and three triangles. The square is found by SubdueGL to be the pattern that when all occurrences of its instances in the graph would be replaced by a single node, the description length of the graph is minimized. SubdueGL also detects that instances of the square in the graph are connected by one edge prompting the inference of recursive production S1. The graph is compressed with S1, and in the second iteration instances of triangles are detected and production S2 is found.

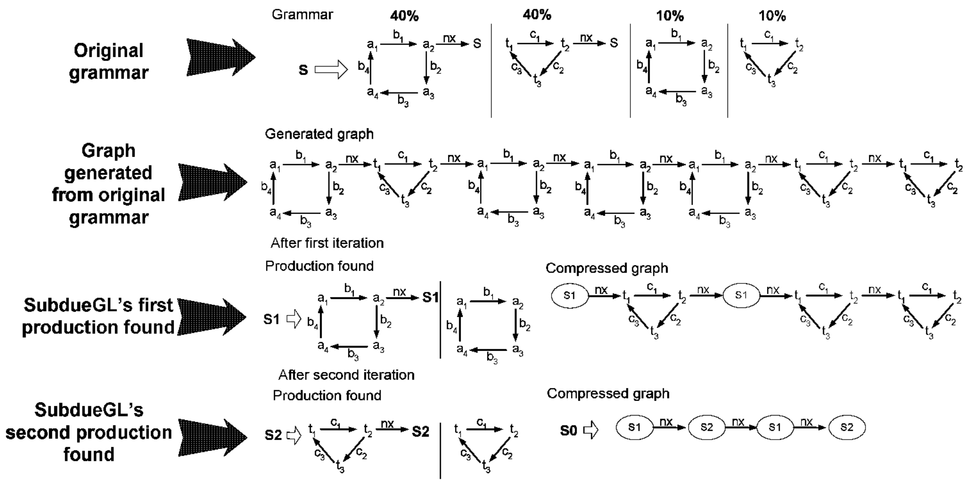

Figure 1. SubdueGL finds recursive grammar in two iterations.

In Figure 2 we generated a tree from a grammar that has two productions. The first production is selected 60% of the time and the second production is a terminal which is a single vertex and is selected 40% of the time. The inferred grammar and the compressed graph are shown on the right side of the figure. We see that the graph grammar inferred by SubdueGL cannot regenerate the tree. It detects only chains of subgraphs connected by an edge.

Our approach grows isomorphic subgraphs similarly as Cook and Holder's[2,3] approach to subgraph discovery with the main difference that it checks for overlap between growing subgraphs. The overlap allows us to propose recursive grammar rules. There are other systems which search for frequent subgraphs in a graph and therefore they could possibly be adopted to graph grammar inference. Kuramochi and Karypis[10] developed FSG. Yan and Han introduced gSpan.[18]

A vast amount of research has been done in string grammar inference.[17] We found only a few studies in graph grammar inference. These approached infer different classes of graph grammars than our approach and therefore we did not do quantitative analysis between them.

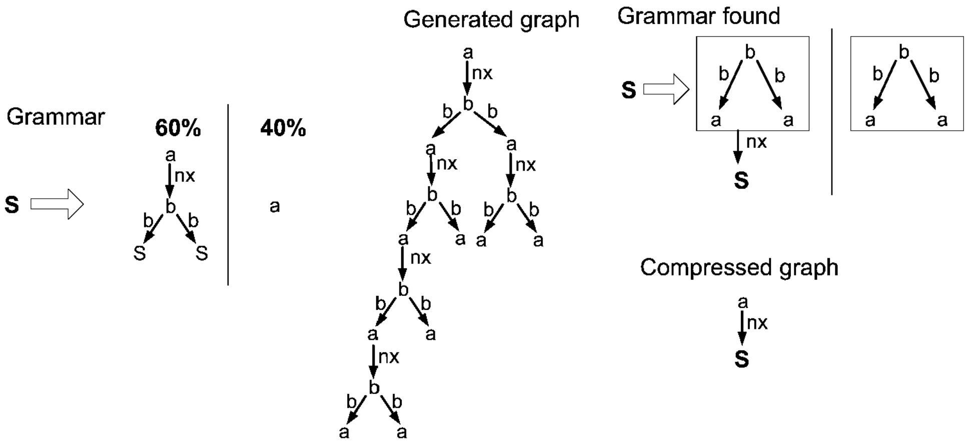

Figure 2. Graph grammar inference from a tree.

We encountered in the existing literature a classification of graph grammars based on the embedding mechanism.[8] 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 3 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. Edge replacement context free recursive graph grammar is the one described in this paper. This grammar is included in the edge replacement context free grammars in Figure 3. The set of grammars inferred by Jonyer *et al.*[6,7] we call chain grammars. Chain grammars are a subset of node replacement recursive graph grammars.

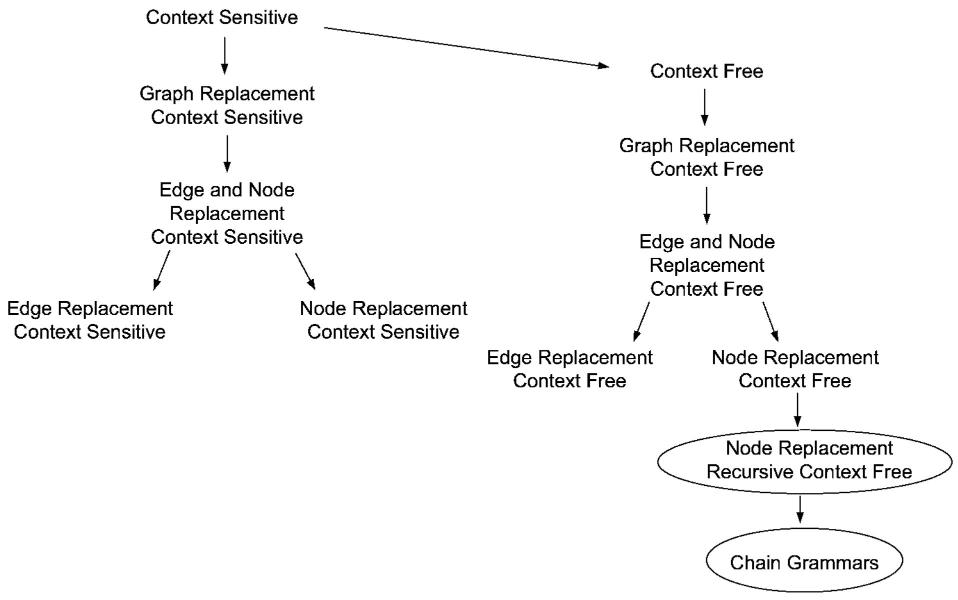
Context Sensitive
Graph Replacement Context Sensitive
Edge and Node Replacement Context Sensitive
Edge Replacement Context Sensitive
Node Replacement Context Sensitive
Context Free
Graph Replacement Context Free
Edge and Node Replacement Context Free
Edge Replacement Context Free
Node Replacement Context Free
Node Replacement Recursive Context Free
Chain Grammars

Figure 3. Hierarchy of graph grammars.

3. Edge Replacement Recursive Graph Grammar

We define a graph as a set of nodes and edges, where each can have a label. Each edge can be directed or undirected. We infer an embedding mechanism for recursive productions which consists of four integers for every non-terminal edge. These integers are node numbers. Two nodes belong to one instance of a graph and two to the other. They describe how instance of a graph defined in the grammar production would be expanded during derivations. In every iteration of the grammar inference algorithm we are finding only one production, and it can be non-recursive or recursive. The reader can refer to examples in Figure 4 and Figure 6 while examining the definition. In Figure 4 (a) we see an example of the grammar used for generation and in Figure 4(b) the equivalent inferred grammar.

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, $\mu : V \rightarrow L$ — is a function assigning labels to the nodes, $\nu : E \rightarrow L$ — is a function assigning labels to the edges, $\eta : E \rightarrow \{0, 1\}$ — is a function assigning direction property to edges (0 if undirected, 1 if directed). L — is a set of labels on nodes and edges.

An edge replacement recursive graph grammar is a 5-tuple $Gr = (\Sigma, \Delta, \Gamma, \Omega, P)$, where Σ — is an alphabet of node labels, Δ — is an alphabet of terminal node labels, $\Delta \subseteq \Sigma$, Γ — is an alphabet of edge labels, Ω — is an alphabet of terminal edge labels, $\Omega \subseteq \Sigma$, P — is a finite set of productions of the form (d, G, C), G is a graph, and there are recursive productions, where $d \in \Gamma - \Omega$, and there is at least one edge in G labeled d. C is an embedding mechanism with a set of connection instructions, $C \subseteq (V \times V; V \times V)$, where V is the set of nodes of G. A connection instruction $(v_i, v_j; v_k, v_l) \in C$ implies that derivation can take place by replacing v_i, v_k in one instance of G with v_j, v_l respectively, in another instance of G. All the edges incident to v_i are incident to v_j, and all the edges incident to v_k are incident to v_l. All the edges incident to v_j and v_k remain unchanged. If, in derivation process after applying connection instruction $(v_i, v_j; v_k, v_l)$, nodes v_i, v_j are adjacent by an edge, we call edge $e = (v_i, v_j)$ a real edge, otherwise edge $e = (v_i, v_j)$ is used only in the specification of the grammar, and we draw it to show two nodes where the connection instructions are applied, and we call this edge a virtual edge.

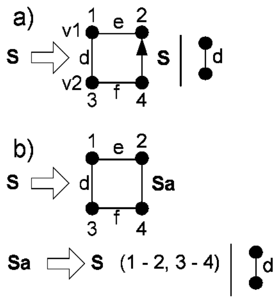

Figure 4. The original grammar (a) used to generate examples and the inferred grammar (b).

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 $(I1, I2, \dots, In)$ 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 $(IR_1, IR_2, \dots, IR_n)$ where every instance IR_k is a subgraph of G. Every instance IR_k consists of one or more isomorphic copies of S_G, overlapping by no more than one vertex in the algorithm for node graph grammar inference and no more than two vertices in edge grammar inference.

In our definition of a 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. The Algorithm

The algorithm operates on a data structure called a substructure. A substructure consists of a graph definition of the repetitive subgraph and its instances. We illustrate it in

Figure 5. Initially, the graph definitions of substructures are single nodes, and there are as many substructure inserted into the queue Q as there are different labels on nodes in the input graph. We expand the substructure in all possible ways by a single edge or by single edge and a node. We allow substructures to grow and their instances to overlap but by no more than two nodes. We evaluate substructures. The total number of substructures considered is determined by the input parameter Limit. The input parameter Beam specifies the width of a beam search, i.e., the length of Q.

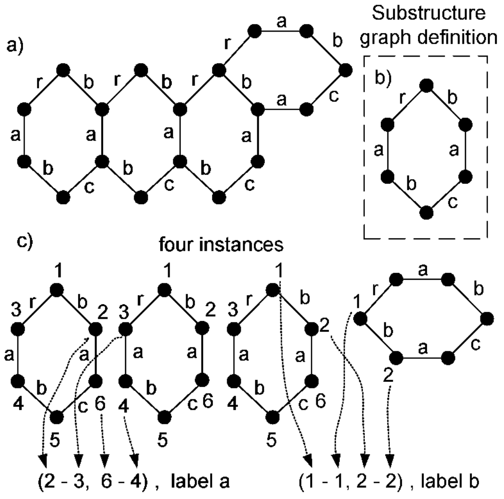

Figure 5. The input graph (a), substructure graph definition (b) and four overlapping instances of repetitive subgraph (c).

If two nodes v_1, v_2 in G both belong to two different instances, we propose a recursive grammar rule. If v_1, v_2 are adjacent by an edge, it is a real edge, and we determine its label which we use to specify the terminating production (see Figure 6). We insert recursive substructures together with non-recursive substructures into the newQ. Recursive substructures compete with non-recursive substructures. They are evaluated with a measure:

$$\frac{size(G)}{size(S) + NT + size(G \mid S)} \tag{1}$$

NT is the number of connection instructions. G|S is a graph G where we compress all instances of the substructure S to a single node. The size is number of nodes plus number of edges. Another successful measure of *size*(*g*) is the Minimum Description Length (MDL) discussed in detail by Cook and Holder.[2] Either of these measures can be used to guide the search and determine the best graph grammar. The algorithm uses a heuristic search whose complexity is polynomial in the size of the input graph to find frequent subgraphs. Checking for overlap between instances of substructures, do not change the complexity of this algorithm.

Algorithm 1 Graph grammar discovery.

```
INFER_GRAMMAR (graph $G$, integer $Beam$, integer $Limit$)
1.  $grammar = \{\}$
2.  repeat
3.    queue $Q = \{v \mid v$ is a node in $G$ having a unique label$\}$
4.    $bestSub$ = first substructure in $Q$
5.      repeat
6.        $newQ = \{\}$
7.        for each substructure $S \in Q$
8.          $newSubs$ = extend substructure $S$ in all
            possible ways by a single edge and a node
9.          $recursiveSub$ = RECURSIFY_SUBSTRUCTURE ($S$)
10.         $newQ = newQ \cup newSubs \cup recursiveSub$
11.         $Limit = Limit-1$
12.         evaluate substructures in $newQ$
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 $Limit \leq 0$
18.     $grammar = grammar \cup bestSub$
19.     $G = G$ compressed by $bestSub$
20. until $bestSub$ cannot compress the graph $G$
21. return $grammar$

RECURSIFY_SUBSTRUCTURE (substructure $S$)
1.  $recS \rightarrow connectList = \{\}$
2.  $recS \rightarrow Instances = \{\}$
3.  for all pairs of instances $(I_1, I_2)$,  $I_1 \in S, I_2 \in S$
4.    if ($I_1$ and $I_2$ overlap on two nodes $v_1, v_2 \in G$)
5.      if ($v_1$, $v_2$ adjacent by an edge in $G$)
6.         edge.type = real, edge.label = label($v_1$, $v_2$) else
7.         $edge.type$ = virtual, $edge.label$ = NULL
8.      $(v_i, v_j; v_k, v_l)$ = GET_CONNEC$(v_1, v_2, I_1, I_2)$
9.      if ($(v_i, v_j; v_k, v_l, edge) \notin (recS \rightarrow connectList)$)
10.   add $(v_i, v_j; v_k, v_l, edge)$ to $(recS \rightarrow connectList)$
11.     if $I_1 \cap IR_k \neq \emptyset$ or $I_2 \cap IR_k \neq \emptyset$, where $IR_k$ is any member of $recS \rightarrow Instances$
12.        $IR_k = IR_k \cup I_1 \cup I_2$ else
13.        create new entry $IR_k = I_1 \cup I_2$ and add it to
           $recS \rightarrow Instances$
return $recS$
```

The algorithm can learn grammars with multiple productions. When we find a production we compress portion of the graph described by the production. Every connected subgraph described by the production is compressed into a node. Then we perform again inference on the compressed graph. We progress with alternating inference and compression until we cannot compress the graph any more.

In Algorithm 1 we represent a substructure by S. We define two functions in Algorithm 1: INFER_GRAMMAR and RECURSIFY_SUBSTRUCTURE. The first function is consistent with Cook *et al.*'s[2] algorithm. Initially, the graph definitions of substructures are single nodes, and there are as many substructures inserted into the queue Q at line 3 as there are different labels on nodes in the input graph. At line 8 we expand the substructure in all possible ways by a single edge or by a single edge and a node. We allow substructures to grow and their instances to overlap but by no more than two nodes. We evaluate substructures at line 12. The total number of substructures considered is determined by the input parameter *Limit*. The input parameter *Beam* specifies the width of a beam search, i.e., the length of *Q*.

Procedure RECURSIFY_SUBSTRUCTURE takes substructure *S* and examines its instances for overlap at line 4. If two nodes in G both belong to two different instances, we propose a recursive grammar rule. In lines 5, 6, and 7 we determine the type of non-terminal edge. If v_1, v_2 are adjacent by an edge, it is a real edge, and we determine its label which we use to specify the terminating production (see Figure 33). Lines 11, 12, and 13 produce recursive instances. Every instance *IR* is a portion of the input graph G which contains two or more overlapping instances of S.

We insert recursive substructures together with non-recursive substructures into the newQ in line 10 of the RECURSIFY_SUBSTRUCTURE procedure. Recursive substructures compete with non-recursive substructures. They are evaluated at line 12. For more details about the algorithm see Refs. 2, 6 and 7.

5. Experiments

5.1. *Methodology*

In our experiments we generate thirty graphs from a known grammar, and then we infer the grammar from every generated graph. We compute the average inference error over these thirty examples. The generated graphs have 40 to 60 nodes. Our generator can assign a random label to a node or an edge. We compare the original grammar and inferred grammar using the following measure of the error:

$$Error = \min\left(1, \frac{\text{matchCost}(g_1, g_2) + \left|\# CI - \# NT\right|}{\text{size}(g_1) + \# NT}\right) \tag{2}$$

where matchCost(g_1, g_2) is the minimal number of operations required to transform g_1 into a graph isomorphic to g_2, or g_2 into a graph isomorphic to g_1. The operations are: insertion of an edge or node, deletion of an edge or node, or substitution of a node or edge label. # *CI* is the number of inferred connection instructions, # *NT* is the number of non-terminal edges in the original grammar, size(g_1) is the sum of the number of nodes and edges in the graph used in the grammar production

matchCost(g_1, g_2) measures the structural difference between two graphs with an algorithm for inexact graph match initially proposed by Bunke and Allermann.[1] For more details see also Refs. 1 and 2. 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 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 by inferred graph structures significantly larger than the graph used in the original grammar. We now describe several experiments showing different aspects of the edge replacement graph grammar inference system.

5.2. *Experiment 1: Virtual and real edges in productions*

In Figure 6 we see the graph on the top where all nodes have the same label and on the bottom of the figure the grammar inferred from this graph. We intend to demonstrate a variety of productions and the nature of edge replacement grammars that our approach can handle. The input graph has four different repetitive patterns. In every pattern subgraphs overlap on two nodes. The part of the graph with overlapping squares is isolated. The rest of the graph is a connected graph. The four patterns correspond to nodes S1, S2, S3, S4 of the first production S. Our approach finds production S last. Production S is a non-recursive node replacement production. We find production S by compressing the input graph with recursive edge replacement productions found earlier. Production S1 we find first because it compresses the graph the most. This production has two non-terminal edges. Edge S1a is virtual. Edge S1b is real. We can replace both S1a and S1b non-terminal edges with the graph on the right hand side of production S1 or terminate. Connection instructions for S1a and S1b are different as is their termination. The terminating edge of S1b is an edge with label q. The termination of S1a is by taking no action. We mark it by two nodes without an edge. We compress to a single node the part of the input graph described by the S1 production before we repeat the inference process. We also do similar compression after finding S2, S3, and S4. The second production we find is S2. This production has two virtual edges as non-terminals. The production S3 has two non-terminal real edges and production S4 has one non-terminal real edge.

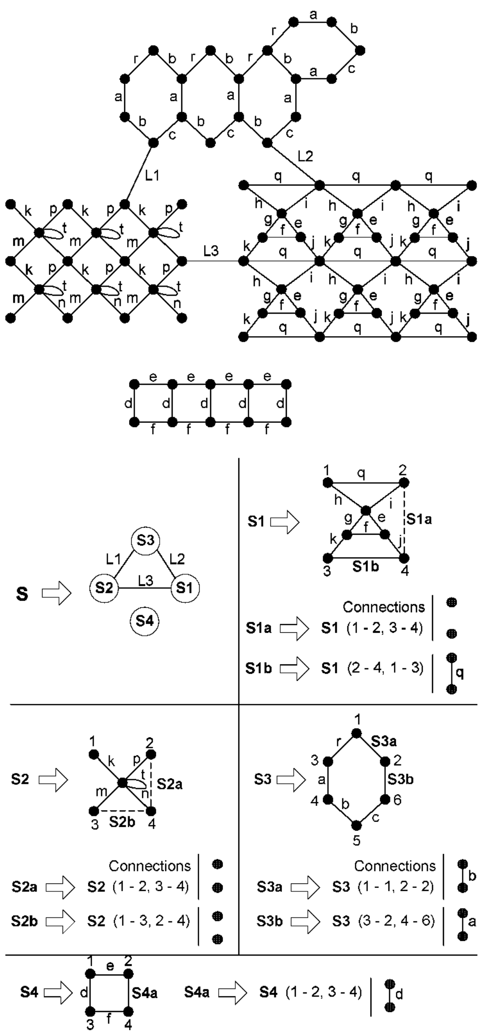

Figure 6. The graph and inferred grammar from this graph.

5.3. ***Experiment 3: Inference error with different graph structures***

We are interested in how inference error depends on grammar structure. We tested several structures. We show results in Figure 7. Every point in the plots in Figure 7 was an average of the inference error from thirty experiments. In every experiment we generated graphs with 40 to 60 nodes. Every label of an edge and a node of the graphs not marked in the Figure 7 and Figure 8 was assigned a label chosen from k distinct labels, where k is an integer from 1 to 7 in Figure 7 and from 1 to 16 in Figure 8. We see that the smallest error we achieved is for the tree structure. As we complicate the structure and increase the average degree of nodes and the ratio of the number of edges to

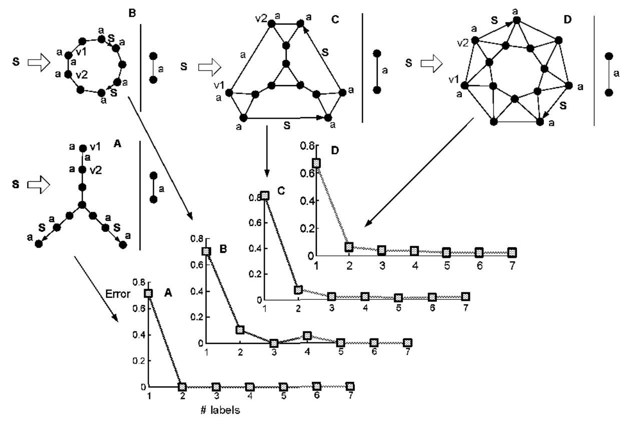

Figure 7. The influence on the error of different graph structures used in grammar productions.

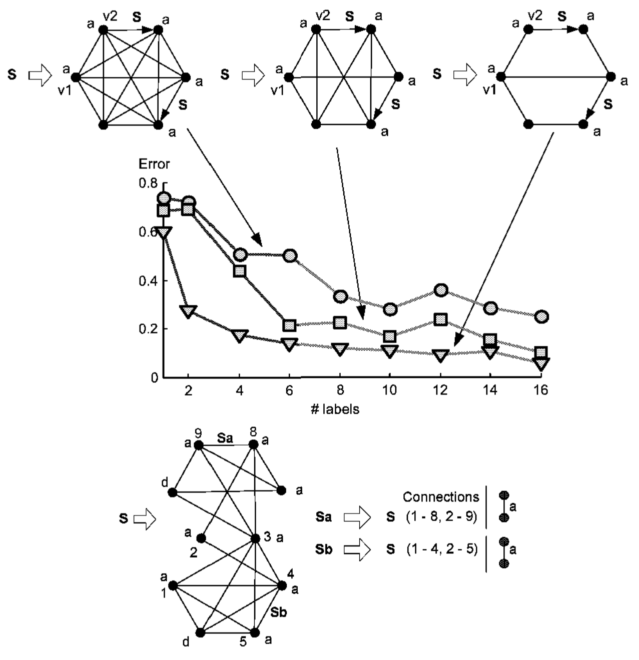

Figure 8. The change in the error with reduced number of edges from the complete graph structure (top) and an example of the inferred grammar (bottom).

the number of nodes, the error increases. The highest error we had with complete graph. We show this case separately in Figure 8. We observed the average value of the inference error for a complete graph with six nodes. Then we removed from the complete graph four edges and repeated the experiment. Next, we remove from the complete graph eight edges and repeated the experiments again. As we see in Figure 8, the more edges we have in the graph and the closer the graph is to the complete graph, the higher the average error. In other words, the closer the graph is to the complete graph the more unique labels we need to decrease the error.

5.4. *Experiment 4: Inference error in the presence of noise*

In Figure 9 we show the results of an experiment where we generated graphs with the number of nodes from 40 to 60. The Peterson graph (Figure 9(a)) was the structure we used in the graph grammar. The Peterson graph has 10 nodes and 15 edges which allowed us to vary the number of non-terminal edges in the structure. We assigned distinct labels to all nodes except six and all edges except six. We generated graphs with 1, 2, 3, 4, and 5 non-terminals and noise value, 0.1, 0.2, … , 0.8. For every value of noise and number of non-terminals we generated thirty graphs from the grammar and computed average inference error over thirty values. We distinguish two types of noise: corrupted and not corrupted. Not corrupted noise is the addition of nodes and edges to the graph structure generated from the grammar. We add the number of nodes equal to (noise/(1-noise))*number_of_nodes and number of edges equal to (noise/(1-noise))*number_of_edges. Every new edge randomly connects two nodes of the graph. We randomly assigned the labels to added edges and nodes from labels already existing in the graph. We do not change the structure generated from the graph grammar in the not-corrupted version. However, in the corrupted version we change the structure of that generated from the grammar graph. After adding additional nodes and edges, in

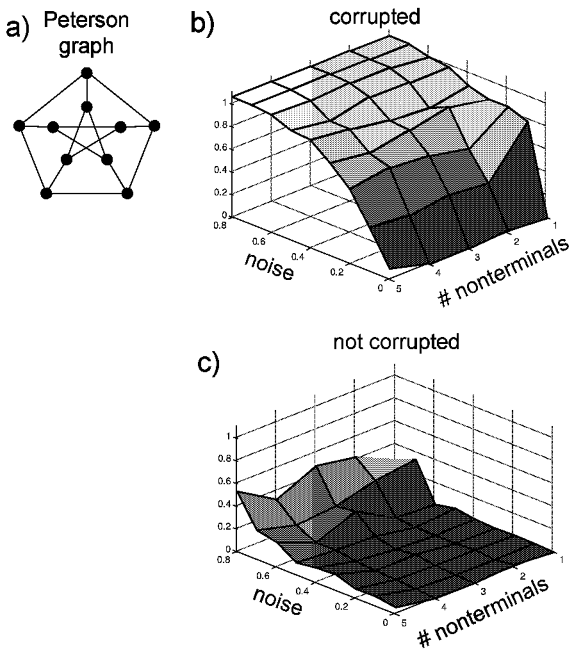

Figure 9. Inference error of a graph grammar with the Peterson graph structure in the presence of noise and different number of non-terminals.

the way we do for non-corrupted version, we redirect randomly selected edges. The number of edges of a graph multiplied by noise gives the number of redirected edges. We randomly assign two new nodes to every selected edge.

The results in Figure 9 show that there is little influence on error from the number of non-terminals. We see an increase in the error in the not-corrupted version when the number of non-terminals reaches 5, but for number of non-terminals 1–4 we do not see any significant changes. Also, the error in the not-corrupted version does not increase significantly as long as the value of noise is less than about 0.5. Corruption of the graph structure, as expected, causes greater error than non-corruption. The error increases significantly even with 0.1 noise, and is close to 100% for noise 0.3 and higher.

5.5. *Experiment 5: Chemical structure*

In Figure 10(a) we show the chemical structure of G tetrad 12. Versions of this structure are used in research on the HIV-1 virus 15. We converted this structure to a graph which we use as an input to our grammar inference system. We found the grammar which represents the repetitive pattern of this chemical structure. We show the grammar in Figure 10(b). This experiment demonstrates the potential application of our approach and also a weakness for further study. Although the grammar production we found

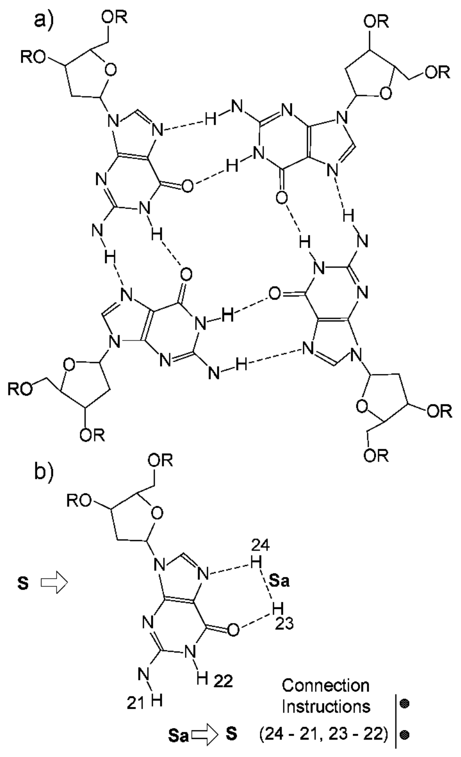

Figure 10. The chemical structure of G tetrad (a) and inferred grammar structure (b).

captures the underlying motifs of the chemical structure, it cannot regenerate the original structure which has the ring form. This limitation results from the fact that our productions use an embedding mechanism that can replicate the structure but is unable to close the ring. If used in generation, our connection instructions connect only two instances that have common edge. One approach that could capture a circle would be looking for sets of instances that form a cycle. We can then propose a cyclic production for a cycle of this specific size (e.g., 4 in experiment 5 data), or a variable-sized cycle.

We also performed experiments with biological networks, XML file structures and other chemical structures, which we will report in other publications. In general, our graph-grammar inference methods have been able to capture known recursive structure in these domains.

6. Conclusions and Future Work

We described an algorithm for inference of edge replacement graph grammars. The performance of the algorithm depends on the number of distinct labels in the input graph. If there is only one label, the algorithm finds a two edge grammar. If we use three or more labels in the input graph, the inference error drops to zero or to a value close to zero in inference of grammars with a graph structure of a tree, cycle, Peterson graph, and tetrahedron. However, as we complicate the structure and increase the average degree of nodes and the ratio of the number of edges to number of nodes, the error increases. The highest error we had is with a complete graph. The closer the graph structure of the grammar is to a complete graph, the more unique labels we need to use in the graph to achieve the same level of average inference error. If we generate graphs from a graph grammar and then add nodes and edges to this graph, it does not influence significantly the inference error in the range of noise 0 to 0.5. There is little influence on error from the number of non-terminal edges in the Peterson graph grammar structure when the number of non-terminals changes from 1 to 4.

In this paper we described the approach to graph grammar inference which extends the class of learnable graph grammars. Node Replacement Recursive Graph Grammar inference was limited to the patterns where instances overlap on exactly one node. In the approach presented in this paper allowing instances to overlap on two nodes led to the definition of real and virtual non-terminal edges. With this approach we can infer the grammar generating chains of squares overlapping on one edge which was not possible with node replacement grammars. Patterns often overlap on two nodes in chemical structures, as we saw in the example of the previous section; therefore, we have a tool which can find and represent important patterns in the chemical domain.

The approach has higher error when inferring more complete graphs. The inferred grammars, as in the example of chemical structure, can represent the underlying pattern of the structure, but cannot regenerate the structure if it has the ring form. The approach requires the existence in the input graph of frequently occurring isomorphic subgraphs and their overlap by one edge to infer recursive productions. Otherwise, the approach can infer non-recursive productions. Eventually, we will integrate inference of non-recursive, node-replacement and edge-replacement productions into one graph-grammar inference system. All these issues represent directions for future research.

References

1. H. Bunke and G. Allermann, Inexact graph matching for structural pattern recognition, in *Pattern Recognition Letters*, 1(4) (1983), pp. 245-253.
2. D. Cook and L. Holder, Substructure Discovery Using Minimum Description Length and Background Knowledge, in *Journal of Artificial Intelligence Research*, Volume 1 (1994), pp. 231-255.
3. D. Cook and L. Holder, Graph-Based Data Mining, in *IEEE Intelligent Systems*, 15(2) (2000), pp 32-41.
4. D. Gernert, Graph Grammars as an Analytical Tool in Physics and Biology. *Biosystems*, Vol. 43, No. 3, (9) (1997), pp. 179-187
5. E. Jeltsch and H. Kreowski, Grammatical Inference Based on Hyperedge Replacement. Graph-Grammars, in *Lecture Notes in Computer Science* 532 (1990), pp. 461-474.
6. I. Jonyer, L. Holder and D. Cook, Concept Formation Using Graph Grammars, in *Proceedings of the KDD Workshop on Multi-Relational Data Mining*, (2002), pp. 71-79.
7. I. Jonyer. L. Holder and. D. Cook, MDL-Based Context-Free Graph Grammar Induction and Applications, in *International Journal of Artificial Intelligence Tools*, Volume 13, No. 1 (2004), pp. 65-79.
8. C. Kim, A hierarchy of eNCE families of graph languages, in *Theoretical Computer Science* 186 (1997), pp. 157-169.
9. J. Kukluk, L. Holder and D. Cook, Inference of Node Replacement Recursive Graph Grammars, in *Sixth SIAM International Conference on Data Mining*, (2006), pp. 544-548.
10. M. Kuramochi and G. Karypis, Frequent subgraph discovery, in *Proceedings of IEEE 2001 International Conference on Data Mining (ICDM '01)*, (2001), pp. 313-320.
11. R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii and U. Alonl, Network Motifs: Simple Building Blocks of Complex Networks, *Science*. Vol. 298, Issue 5594, (2002) pp. 824-827.
12. S. Neidle (editor), *Oxford Handbook of Nucleic Acid Structure* (Oxford University Press, 1999), pp. 326.
13. G. Nevill-Manning and H. Witten, Identifying hierarchical structure in sequences: A linear-time algorithm, in *Journal of Artificial Intelligence Research*, Vol. 7 (1997), 67-82.
14. G. Palla, I. Derényi, I. Farkas and T. Vicsek, Uncovering the overlapping community structure of complex networks in nature and society, *Nature* 435 (2005), pp. 814-818.
15. A. Phan, V. Kuryavyi, J. Ma, A. Faure, M. Andreola and D. Patel, An interlocked dimeric parallel-stranded DNA quadruplex: A potent inhibitor of HIV-1 integrase, in *Proc. Natl. Acad. Sci*, (2005) 102, pp. 634 – 639.
16. T. Oates, S. Doshi and F. Huang, Estimating maximum likelihood parameters for stochastic context-free graph grammars, in *Lecture Notes in Artificial Intelligence* 2835 (Springer-Verlag, 2003), 281-298.
17. Y. Sakakibara, Recent advances of grammatical inference. *Theoretical Computer Science*, 185 (1997) 15-45.
18. X. Yan and J. Han, gSpan: Graph-based substructure pattern mining, In *IEEE International Conference on Data Mining* (Maebashi City, Japan, 2000), pp. 51-58.