INFERENCE OF NODE AND EDGE REPLACEMENT GRAPH GRAMMARS

by

JACEK KUKLUK

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2007

ACKNOWLEDGEMENTS

I would like to thank Dr. Lawrence B. Holder for leading this research, for his advice, patience, and encouragement.

October 23, 2006

ABSTRACT

INFERENCE OF NODE AND EDGE REPLACEMENT GRAPH GRAMMARS

Publication No. ______

Jacek Kukluk, PhD.

The University of Texas at Arlington, 2007

Supervising Professor: Lawrence B. Holder

In this dissertation we study the inference of node and edge replacement graph grammars. The 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 graph grammar production. If subgraphs overlap by two nodes or two nodes and an edge, we propose an edge replacement graph grammar production. We also can infer a hierarchy of productions by compressing portions of a graph described by a production and then inferring new productions on the compressed graph.

We validate the approach to node replacement grammar inference in experiments where we generate graphs from known grammars and measure how well the approach infers the original grammar from the generated graph. We show how this method performs in extracting the organization of XML files. We convert an XML file into a tree and infer a graph grammar from it. We compare the inferred graph grammar to the Document Type Definition of the XML file. We report the graph grammar we found from XML files used in the National Library of Medicine, the United States Patent and Trademark Office, and major baseball leagues. We also apply the algorithm to biological domains. We show the graph grammars found in biological molecules and in biological networks, and analyze learning curves of the algorithm as we increase the number of biological networks input to the method.

We also describe an algorithm and experiments for inference of edge replacement graph grammars. This method generates candidate recursive graph grammar productions based on finding 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 using the Minimum Description Length (MDL) of a graph and the size of a graph. 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.

We briefly discuss other grammar inference algorithms indicating that our study extends classes of learnable graph grammars.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figure Page

LIST OF TABLES

Table Page

CHAPTER 1

INTRODUCTION

1.1 Overview

Noam Chomsky [Chomsky56] pointed out that one of the main concerns of a linguist is to discover simple grammars for natural languages and study those grammars with the hope of finding a general theory of linguistic structure. A vast amount of research has been done in inferring grammars [ICGI94 , ICGI98, ICGI00, ICGI02, Lari91]. These analyses focus on string grammars where symbols appear in a sequence. We are concerned with graph grammars, which can represent much larger classes of problems than string grammars. We examine the classes of graph grammars presently learnable. As string grammars represent the language, we are looking for graph grammars that represent graph properties and can generalize these properties from finite graph examples into generators that can generate an infinite number of graphs. String grammars can be inferred from a finite number of sentences and generalize to an infinite number of sentences. Inferring graph grammars will generalize the knowledge from the examples into a concise form and generalize to an infinite number of entities from the domain. We examine existing approaches to the inference of graph grammars and experimentally explore one of them showing its features and limitations.

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 [Gernert97, Milo02]. We study the problem of grammar inference. We introduce an algorithm which builds on previous work in discovering frequent subgraphs in a graph [Cook94]. 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 developed an algorithm for edge replacement grammar inference where we check for overlap by two nodes. If an overlap exists and there is an edge between the overlapping nodes, we propose a grammar production with a real non-terminal edge. If there is no edge between overlapping nodes, we propose a production with a virtual edge. We can apply grammar inference algorithms in domains with relational data. In this work we selected several of them: chemical structures, XML files, and biological networks. A vast amount of research has been done in string grammar inference [Sakakibara97]. We found only a few studies in graph grammar inference.

1.2 Contributions

We developed two algorithms to infer graph grammars from structured data represented as a graph. The methods can detect recursive and non-recursive motifs in structured data and build a hierarchy of recursive graph grammar productions. We showed how the grammars we can infer fit into the hierarchy of graph grammars. There was no method shown before which can infer these classes of graph grammars. In this work we use graph grammar as a data mining tool. We often refer to the ability of a graph grammar to regenerate the input graph, but inferring such grammars is not the purpose of the work. Graph grammars in our study show interesting patterns and organize data into a hierarchy. We implemented the algorithms and tested them on synthetic and non-synthetic data. For this reason we developed and implemented a generator which generates a graph from a known graph grammar. We showed how inferring graph grammars depends on the presence of noise, the complexity of graph grammar structure, the number of different labels present in the graph, and the size of the generated graph. We showed limitations of node replacement recursive graph grammars and how edge replacement recursive graph grammars overcome some of these limitations. To validate the approach, we conducted experiments in real-world domains: chemical molecules, biological and XML file structure. We inferred the structure of XML files which describe data in the National Library of Medicine, the United States Patent and Trademark Office, and in the domain of baseball. We show how the algorithms perform in inferring grammars from biological networks and how in this domain inference error depends on the number of examples in the input set. We experimentally verified the polynomial complexity of the algorithm in this domain.

Chapter two describes related work in graph grammar inference algorithms and frequent subgraph discovery algorithms. In chapter three we describe classes of graph grammars and give the definition of a node and an edge replacement grammar our algorithm is capable to infer showing their place in the hierarchy of graph grammars. Chapter four presents the algorithms for node and edge replacement graph grammar inference. Chapter five describes the graph generator used in the experimental evaluation. Experiments with node replacement graph grammars we placed in chapter six and experiments with edge replacement graph grammars in chapter seven. Chapter eight presents experimental results for inferring XML schema using graph grammars. Chapter nine contains learning curves and experiments with biological networks. Chapter ten closes the dissertation with conclusions and future directions.

CHAPTER 2

RELATED WORK

In this chapter we describe existing approaches to graph grammar inference. These ideas are based on decomposition of a graph, inference of probabilities associated with grammar rules, learning deterministic tree grammars, and searching for an edge connecting frequent subgraphs. The second part of this chapter discusses approaches to frequent subgraph discovery.

2.1 Graph Grammar Inference

Jeltsch and Kreowski [Jeltsch90] 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 (Jeltsch, 1990) 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. In the example in Figure 1 the initial grammar has productions S-> $K_{3,1}$, S-> $K_{3,2}$, S-> $K_{3,3}$, S-> $K_{3,4}$. The production S-> $K_{3,2}$ is decomposed into two productions. If in the second production D-> $K_{3,1}$, we rename D with S, this production and initial production S-> $K_{3,1}$ are the same and can be reduced with the REDUCE operation.

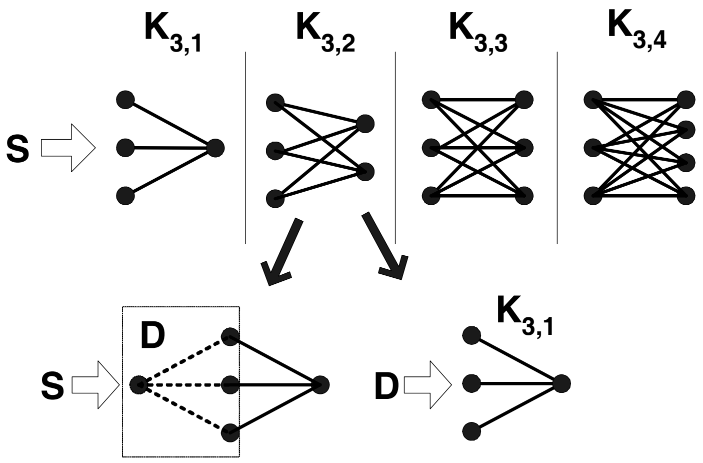


Figure 1: Initial grammar and decomposition of productionS-> $K_{3,2}$ into two productions.

Oates et al. [Oates03] 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 $p(E| S, \theta)$. Their work is based on the work of Lari and Young [Lari91] in estimation of stochastic-context free string grammars. In the example in Figure 2, the grammar has three rules. In their experiments they generated 10 graphs from this grammar with initial probabilities 0.6, 0.2, 0.2 associated with first, second and third rule respectively. With their algorithm they estimated probabilities and found their values θ=(0.6486, 0.2973, 0.0542) which are close to the initial values.

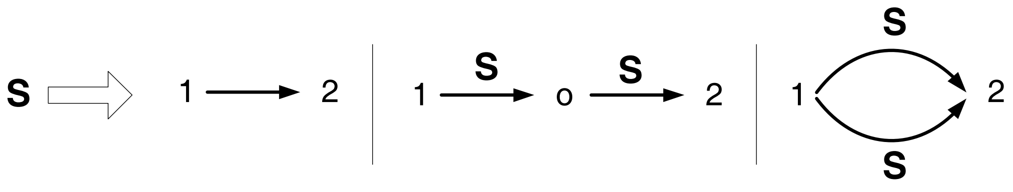


Figure 2: Grammar used in Oates et al. [Oates03] experiments to infer probabilities associated with three grammar rules.

In terms of similarity to string grammar inference we consider the Sequitur system developed by Nevill-Manning and Witten [Nevill97]. Sequitur infers a hierarchical structure by replacing substrings based on grammar rules. The new, compressed string is searched for substrings which can be described by the 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.

Carrasco et al. [Carrasco01] gave an algorithm for learning deterministic tree grammars. Their approach emerges from concepts of learning context free sting grammars. The problem was previously studied by Sakakibara [Sakakibara92]. Carrasco et al. says that the problem of identifying context free grammars is equivalent to the problem of identifying regular tree languages. It is because of the derivation trees of context free grammars form a regular tree language. A deterministic tree automata DTA is an acceptor of rooted, ordered directed trees. The set of accepted trees defines regular tree language. DTA is a 4-tuple A=(Q, V, δ, F), where

-Q is a finite set of states;

-V is a finite set of labels;

- $F \subset Q$ is the subset of accepting states

- $\delta = \{ \delta_0, \delta_1, \ldots, \delta_n \}$ is a set of transition functions of the form $\delta_k: V \rightarrow Q^k$

Trees are represented by a functional notation. Following the example provided by Carrasco et al. [Carrasco01], the tree in Figure 3 is represented by t=a (b (a (b c)) c. The root has labeled ‘a’ and it branches into two subtrees t_1 = (b (a (b c) and t_2=c. In general, a tree at a node with label x can branch into subtrees $t_1, t_2, \ldots, t_n$. The tree or a subtree is represented by $t=x(t_1, t_2, \ldots, t_n)$. DTA processes the tree bottom-up. It results from recursive definition of the transition function:

$$\delta(t) = \begin{cases} \delta_k(x,\ \delta(t_1), \ldots, \delta(t_k)) & \textit{if } t \textit{ is a subtree} \\ \delta_0(a) & \textit{if } t \textit{ is a leaf} \end{cases}$$

Every node in t has an associated state in DTA. In order to process the tree in Figure 3, the following transition functions are defined δ_0 (b)=q_1 , δ_0(c)=q_2 , δ_2 (a, q_1 , q_2)=q_2 , δ_1 (b, q_2)=q_1. The tree is processed from the bottom starting with b and c nodes. With transition δ_0 they are assigned states q_1 and q_2. Following up the tree we read 'a' and assigned state q_2 with transition δ_2. Next δ_1 (b, q_2)=q_1 and finally the root of the tree is found to be in state q_2. The state assigned to the root must be an accepting state, for the tree to be accepted by DTA. If V^T is the set of all trees whose nodes are labeled with symbols from V, DTA accepts regular tree language $L = \left\{t \in V^T : \delta(t) \in F\right\}$. Carrasco et al. inference algorithm is looking for states represented by subtrees and transition functions δ. They are also looking for probabilities of every transition to infer stochastic tree automata.

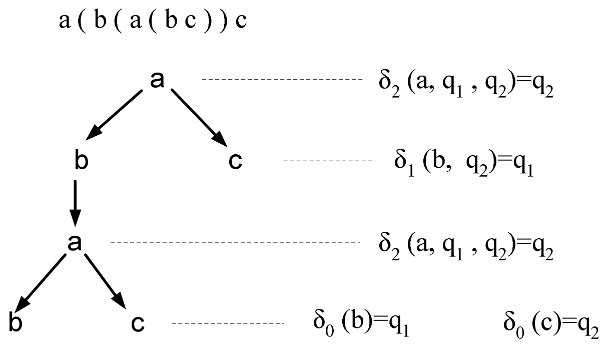

a (b (a (b c)) c
a
b
c
a
b
c
δ_2 (a, q_1 , q_2)=q_2
δ_1 (b, q_2)=q_1
δ_2 (a, q_1 , q_2)=q_2
δ_0 (b)=q_1
δ_0 (c)=q_2

Figure 3: The ordered, rooted tree coded as a(b(a(bc))c and processed bottom-up by deterministic tree automata [Carrasco01] .

The most relevant work to this research is Jonyer et al.'s approach to node-replacement graph grammar inference [Jonyer02, Jonyer04]. 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→ 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. 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 4. In this figure, above the productions, we indicated a percentage that signifies the probability with which we are using every production. In Figure 4 we show a grammar which 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 nonterminal square and nonterminal triangle. Terminal square and triangle are assigned probability 10%. In Figure 4, below the grammar we drew one generated graph. It contains four squares and three triangles. Square is found by SubdueGL to be the pattern that when all occurrences of its instances in the graph would be replaced by single node, the description length of the graph is minimized. SubdueGL also detects that instances of 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.

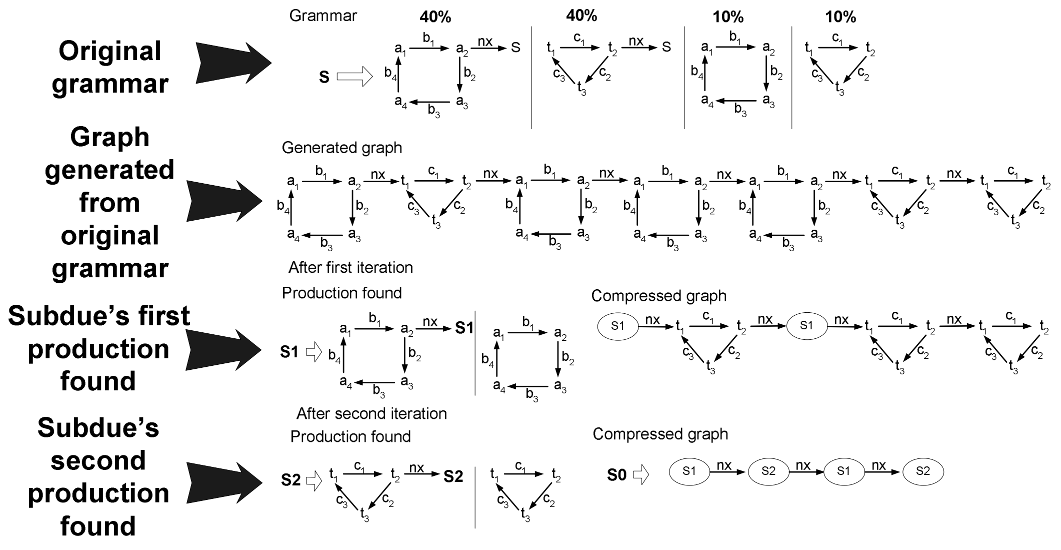


Figure 4: SubdueGL finds recursive grammar in two iterations.

In Figure 5 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. and the inferred grammar and the compressed graph are shown on the right side of the figure. We see that inferred by SubdueGL graph grammar cannot regenerate the tree. It detects only chain of subgraphs connected by an edge. We see later in the dissertation graph grammars inferred with node replacement grammars which can regenerate trees. Next, we describe frequent subgraph discovery algorithms.

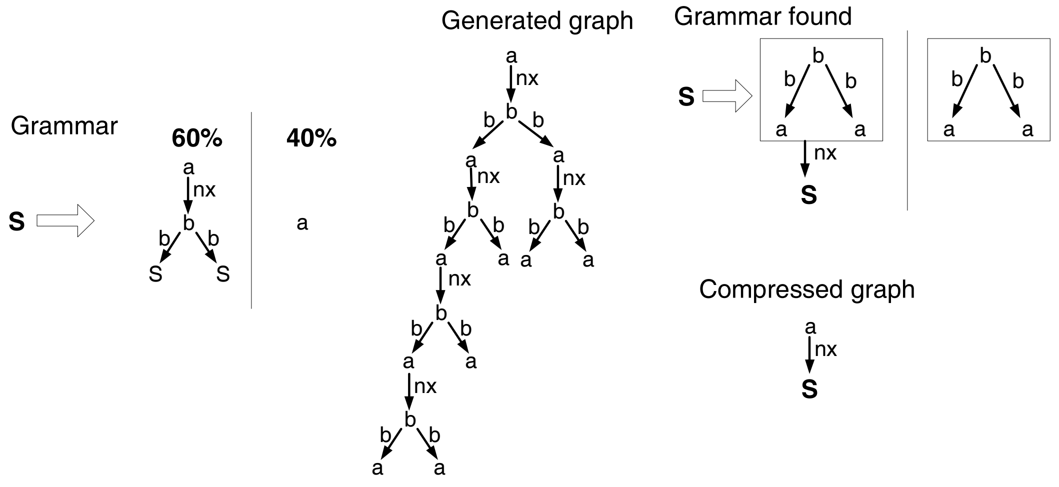


Figure 5: Graph grammar inference from a tree.

2.2 Frequent Subgraph Discovery Algorithms

In our approach we use the frequent subgraph discovery algorithm Subdue developed by Cook and Holder [Cook94]. We would like to mention other algorithms developed to discover frequent subgraphs and therefore have the potential to be modified into algorithms which can infer a graph grammar. Kuramochi and Karypis [Kuramochi01] implemented the FSG algorithm 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 [Yan02], 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.

The challenge of using frequent subgraph mining algorithms like gSpan or FSG to infer graph grammars would be the modification to allow subgraph instances to overlap. Overlapping substructures are available as an option in the Subdue algorithm [Cook94]. 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.

We gave an outline of several approaches to inference of graph grammars. In addition to the described approaches we would like to mention a few others. Doshi at el. [Doshi02] similar to SubdueGL use instances of frequent subgraphs that minimize description length of a graph to infer stochastic graph grammars. Fletcher [Fletcher01] describes an algorithm to learn graph grammars which represent two dimensional structures drawn on a discrete Cartesian grid. Sánchez at el. discus inference of graph grammars that describe texture symbols [Sanchez01]. We found that only a few papers address inference of graph grammars. The problem is open for more systematic study.

Described approaches differ in the way the graph grammar is built. The approach of Jeltsch and Kreowski [Kreowski90] starts from the grammar where all the sample graphs are initially on the right hand side of the production. The algorithm progresses by transformation towards reduction and generalization. SubdueGL works in the opposite direction. It starts with no grammar and then discovers one production in the first iteration, compresses the graph with it and subsequent iterations bring more productions. Oates et al.'s approach builds an automaton that accepts ordered trees. They are looking for states, transition functions and probabilities of productions.

Jeltsch and Kreowski's approach requires a set of disconnected graphs as an input. SubdueGL can find recursive graph grammas in a set of disconnected graphs as well as in one connected graph. Figure 4 illustrates an example where SubdueGL finds two productions of the recursive graph grammar. These productions together with the compressed graph can regenerate the original graph. SubdueGL finds chains of isomorphic subgraphs connected by one edge and replaces them with a production rule. This feature is also a major limitation of the algorithm. In Figure 5 we used a grammar that can generate trees. We used one of the generated trees as an input to SubdueGL. We show the grammar found by SubdueGL on the right side of the drawing. SubdueGL found the chain of isomorphic subgraphs in the tree, but the limitation that subgraphs are connected by one edge does not allow for learning a grammar which represents a tree. The learned productions together with the compressed graph cannot regenerate the input tree.

We conclude that research in graph grammar inference has focused on classes of graph grammars limited to a subset of context free graph grammars operating on:

- chains of isomorphic subgraphs
- rooted ordered trees,
- undirected unlabeled graphs for which operations of decomposition and merging into a grammar can be defined

Based on analyzed research in the inference of graph grammars we would like to list three directions:

(1) Decomposition of sample graphs and merging a graph grammar from decomposed subgraphs. This approach is inspired by Jeltsch and Kreowski's theoretical study. It involves decomposition of sample graphs into smaller units and merging them into a graph grammar. Other methods of decomposing and merging are open directions.

(2) Building automata which accept sample graphs. The automata theory and languages for strings is a very well defined field [Hopcroft79]. String grammar inference is also a well studied problem [ICGI00,ICGI94,ICGI98,ICGI02]. The concepts of automata inference that accept graphs can draw from these studies. Carrasco et al. [Carrasco01] studied rooted, ordered trees which were coded into strings. The string was processed by deterministic tree automata to find if the tree represented by it is accepted in the language. Representing other classes of graphs as strings would allow for applying concepts from string processing into processing graphs.

(3) Finding frequently occurring instances of subgraphs and examination of their connections. SubdueGL examines if instances of the same subgraph are connected by one edge. The one edge restriction is a major limitation of this approach. An open research problem is in finding methods to examine connections between frequent subgraphs which are more complicated than one edge connection. Consequently, how to construct a graph grammar from frequent instances of subgraphs and information about their connections becomes an issue we study in this dissertation. In the next chapter we discuss definitions and classes of these graph grammars.

CHAPTER 3

CLASSES OF GRAPH GRAMMARS

In this chapter we introduce classes of graph grammars as they are known in the literature. We describe the hierarchy of graph grammars and define classes of graph grammars that we can infer with our algorithm: node and edge replacement graph grammars.

3.1 Graph Grammar Definition

The Chomsky hierarchy of text grammars includes four types: type 3 - regular grammars, type 2 -context-free grammars, type 1 - context-sensitive grammars, and type 0 - unrestricted grammars. The Chomsky hierarchy of string grammars is well-defined because of its relation to finite state automata, pushdown automata, and Turing machines. Graph grammars do not have such relations.

In this section we give an overview of different classes of graph grammars. These classes are often investigated in the literature. In this survey of definitions and hierarchy of graph grammars we intend to provide an initial framework for the graph grammar inference problem.

A *graph grammar* is a pair G = (S, P), where S is the starting graph and P is a set of production rules. *Graph replacement grammars* have productions of the form (M, D, C), where M is the mother graph, D is the daughter graph, and C is a set of connection instructions [Rozenberg97]. All occurrences of subgraph M in a host graph

H are replaced with D using the set of connections C. L(G) is the set of graphs generated by graph grammar G.

Defining a new class of graph grammars requires specification of rewriting rulers (M and D), and an embedding mechanism C. The embedding mechanism is the criterion to classify graph grammars. Authors also point out that embedding mechanisms are among the most important features in studies about graph grammars [Janssens83 , Flasinski98].

We can distinguish two classes of grammars based on the nature of productions: node replacement graph grammars and edge replacement graph grammars. Janssens and Rozenberg [Janssens80] introduce node label controlled (NLC) graph grammars.

3.2 Node Label Controlled (NLC) Graph Grammar

A *node label controlled graph grammar* (NLC) [Rozenberg97] is a 5- tuple $G = (\Sigma, \Delta, P, C, S)$, where

Σ - is an alphabet of node labels,

Δ - is an alphabet of terminal node labels, $\Delta \subseteq \Sigma$,

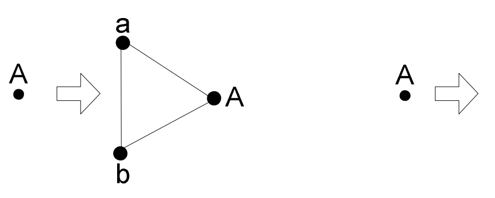
P -is a finite set of productions which are pairs (d, Y), with $d \in \Sigma$ and Y is a graph

C -is a connection relation, a function from Σ into 2^{Σ}

S - is the initial graph

NLC productions replace a single vertex with a graph. There is no separate connection relation for every production. The connection relations are global, the same for all productions. The connection relations apply only to neighboring vertices of a mother vertex. The neighborhood of a vertex *v* in a graph is the set of all the vertices adjacent to *v*. If the mother vertex does not have neighbor vertices labeled as in the list of connection relations C, then these connections are not introduced. Because the embedding of a daughter graph D depends on the labels on the neighboring vertices of the mother vertex M, some papers [Rozenberg86] point out the context-sensitive nature of NLC embedding. We observe that the labels of the vertices adjacent to the mother vertex M do not influence the decision of whether M will be replaced by D or not. Those labels decide only the nature of the connections of D to the host graph. Therefore, we classify NLC grammars to context free grammars.

As an example [Rozenberg86], let $\Delta = \{a,b\}$, $\sum = \{A,a,b\}$, and S be a node with label A, then the two productions of P are given below and the connection relation C is on the right of the productions.



$$C = \begin{Bmatrix} A, & \{a,b\} \\ a, & b \\ b, & a \end{Bmatrix}$$

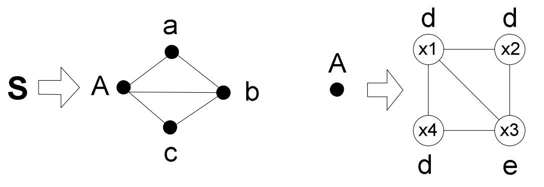
Figure 6: Example of Node Label Controlled (NLC) Graph Grammar.

The language of the graphs generated by the above grammar consists of all bipartite graphs $K_{n,n}$. Neighborhood controlled embedding (NCE) graph grammars have a similar definition to NLC grammars. The difference is that the embedding relation C maps labels of neighboring vertices of M to particular vertices (not based on labels) of a daughter graph D. The embedding relation is no longer global but is given for every production separately.

3.3 Neighborhood Controlled Embedding (NCE) Graph Grammar

The *neighborhood controlled embedding* (NCE) graph grammar [Rozenberg97] is a 4-tuple $G = (\Sigma, \Delta, P, S)$, where Σ, Δ, and S are defined as in the definition of NLC grammars. P is a finite set of productions, which are pairs (d, Y, C), with $d \in \Sigma$ and Y is a graph. C is a connection relation, $C \subseteq \Sigma \times V_D$, and V_D is a set of nodes of Y.

For example the initial graph S and a production with connection relation of an NCE grammar are given below. Labels a, b, c of a host graph and vertices x1, x2, x3, x4 are used in specifying the connection relation C.



$$C = \left\{ \begin{matrix} a, & x1 \\ b, & \{x2, x3\} \\ c, & x4 \end{matrix} \right\}$$

Figure 7: Example of Neighborhood Controlled Embedding (NCE) Graph Grammar.

Janssens and Rozenberg [Janssens82] proved that the set of graphs generated by NCE grammars is the same as generated by NLC grammars (NCE=NLC). edNCE has more generation power than NCE and NLC. The small letter ‘e’ before the name of the grammar means that edges are labeled and they are used in connection instructions. The ‘d’ means that edges are directed.

3.4 Edge-Labeled Directed Neighborhood Controlled Embedding (edNCE) Graph Grammar

An *edge-labeled directed neighborhood controlled embedding (edNCE) graph grammar* is a system $G = (\Sigma, \Delta, \Gamma, \Omega, P, C, S)$, 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,

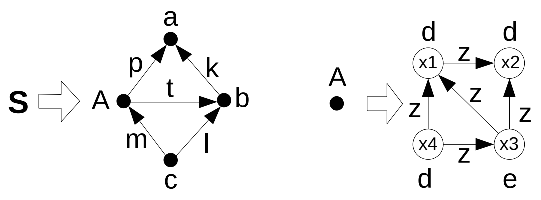
Ω -is an alphabet of terminal edge labels, $\Omega \subseteq \Gamma$,

P -is a finite set of productions of the form (d, Y, C), with $d \in \Sigma$ and Y is a graph,

C -is a connection relation, $C \subseteq \Sigma \times \Gamma \times \Gamma \times V_Y \times \{in, out\}$,

S - is the initial graph.

For example the connection relation given below specifies how vertices of a daughter graph x1, x2, x3 will be connected to the vertices of a host graph. The first line from the list of connection relations means that vertex x1 of the daughter graph will be adjacent to a vertex with label 'a' of a host graph if vertex with label 'a' is adjacent to the mother vertex 'A' on an edge labeled 'p'. The label on an edge between x1 and 'a' will be 'd'. The direction of this edge will be from x1 to 'a'. Remaining lines have similar meaning.



$$C = \left\{ \begin{matrix} a, p, d, x1, in \\ b, t, f, x2, in \\ c, m, g, x3, in \end{matrix} \right\}$$

Figure 8: Example of Edge-Labeled Directed Neighborhood Controlled Embedding (edNCE) Graph Grammar

3.5 Hierarchy and Other Classes of Node Replacement Graph Grammars

We gave the definition of an edNCE grammar. The eNLC and edNLC grammars also were defined [Rozenberg97, Flasinski98, Kim97]. Definitions of the eNLC and edNLC grammars used the connection relation based on labels on vertices of mother and daughter graphs and labels on edges. We can impose restrictions on connection relations of NLC, NCE, eNLC, eNCE grammars to form new classes of grammars. The desired property of these classes is confluence. A graph grammar is *confluent* if the result of a derivation does not depend on the order of production applied. If the result depends on the order of derivation, the grammar is *not confluent*. We list below some of the restrictions [Rozenberg97, Flasinski98, Kim97]. In parentheses we give an abbreviation used as a prefix in the names of grammars.

(B) *Boundary graph grammars.* No two non-terminal nodes are adjacent in the right-hand side of each production and in the starting graph. Boundary grammars are confluent.

(Lin) *Linear graph grammars*. At each derivational step, daughter graph contains at most one non-terminal label.

(A) *Apex graph grammars*. Connection instructions contain only terminal nodes.

(-) *Regular graph grammars.* The right hand side graph is a single non-terminal or consists of two connected nodes, terminal and non-terminal.

To show the relation of different node replacement grammars we reproduced the results of Changwook Kim [Kim97] in Figure 9. The arrows indicate inclusion. For instance, the arrow from B-NCE to NCE means that B-NCE $\subseteq$ NCE.

$$\begin{tikzcd}
 & \text{NLC=NCE} \arrow[r] & \text{eNCE} & \\
 & \text{B-NCE} \arrow[u] \arrow[r] & \text{B-eNCE} \arrow[u] & \\
 & \text{B-NLC} \arrow[u] & & \\
\text{Lin-NCE} \arrow[ur] \arrow[r] & \text{Lin-eNCE} \arrow[u] \arrow[uur] & \text{A-NCE} \arrow[uul] \arrow[r] & \text{A-eNCE} \\
\text{Lin-NLC} \arrow[u] & & \text{A-NLC} \arrow[u] \arrow[uul] & \\
 & \text{Lin-A-NCE} \arrow[uul] \arrow[uur] \arrow[r] & \text{Lin-A-eNCE} \arrow[uul] \arrow[uur] & \\
 & \text{Lin-A-NLC} \arrow[uul] \arrow[u] \arrow[uur] & &
\end{tikzcd}$$

Figure 9: A hierarchy of node replacement graph grammars [Kim97].

Hyperedge replacement grammars is another class of graph grammars. Their definition [Rozenberg97, Habel92, Drewes90] allows for replacement of more complex structures than a single edge. Our focus is on inference of graph grammars and for this purpose we are interested in a grammar, which uses only one edge in its productions. An *edge replacement graph grammar* is a system $G = (\Sigma, \Delta, \Gamma, \Omega, P, C, S)$, where $\Sigma, \Delta, \Gamma, \Omega, S$ are as defined before, P is a finite set of productions of the form (e, Y, C), e is a single edge with a label from $(\Gamma - \Omega)$, Y is a graph, and C is a gluing relation, $\begin{Bmatrix} head(e) & \rightarrow & begin(Y) \\ tail(e) & \rightarrow & end(Y) \end{Bmatrix}$.

In every production, the right hand side, graph *Y*, has two nodes marked in addition to their labels *begin* and *end*. When the production is applied, edge *e* is removed from the host graph and vertices incident to *e* are replaced with Y's vertices marked *begin* and *end*. Vertices of the host graph previously incident to *e* preserve all other connections to the host graph. The labels of these two vertices are replaced by new labels of graph Y.

Until this point we described different classes of graph grammars where the decision of whether a production will be applied or not did not depend on the environment of the replaced edge, node, or graph. All these classes of grammars we classify to context free graph grammars. We would like to introduce two definitions of context sensitive grammars.

Edge replacement context sensitive graph grammar is a system $G = (\Sigma, \Delta, \Gamma, \Omega, P, C, S)$, where

$\Sigma, \Delta, \Gamma, \Omega$, and S are as defined before,

P - is a finite set of conditional productions of the form

$$if\ \begin{Bmatrix} neighborhood(head(e)) \subseteq O\ and \\ neighborhood(tail(e)) \subseteq P \end{Bmatrix} than\ \ e \to Y\ and\ C = \begin{Bmatrix} head(e) & \to & begin(Y) \\ tail(e) & \to & end(Y) \end{Bmatrix}$$

e is a single edge with a label from $(\Gamma - \Omega)$ and Y is a graph. Replacement of an edge e with a graph Y occurs only if specified conditions are met.

C -is a gluing relation, where O and P are sets of labels $O,\ P \subseteq \Sigma$.

A *context sensitive (edNCE) graph grammar* is a system

$G = (\Sigma, \Delta, \Gamma, \Omega, P, C, S)$, where

Σ, Δ, Γ, Ω, and S are as defined before,

P -is a finite set of conditional productions of the form

$$if\ \{neighborhood(d) \subseteq O\}\, then\ d \to Y\ \ and\ C \subseteq \Sigma \times \Gamma \times \Gamma \times V_Y \times \{in, out\},$$

with $d \in \Sigma$ and Y is a graph, C - is a connection relation, O is a set of labels $O \subseteq \Sigma$.

Replacement of a vertex d with a graph Y occurs only if conditions specified are met.

3.6 Node Replacement Recursive Graph Grammar Definition

We give the definition of a graph and a graph grammar which is relevant to our approach. The defined graph has labels on vertices and edges. Every edge of the graph can be directed or undirected. The definition of a graph grammar describes the class of grammars that can be inferred by our approach. 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. This 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 dissertation.

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.

A *node replacement recursive graph grammar* is a tuple $Gr = (\Sigma, \Delta, \Gamma, P)$, 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, which are all terminals,

P - is a finite set of productions of the form (d, G, C), where $d \in \Sigma - \Delta$, G is a graph, and there are two types of productions:

(1) *recursive productions* of the form (d, G, C), with $d \in \Sigma - \Delta$, 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 algorithm does not infer an embedding mechanism for these productions).

3.7 Edge Replacement Recursive Graph Grammar Definition

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 two types of productions:

(1) *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 call this edge a *virtual edge*.

(2) *non-recursive production,* where $d \in \Sigma - \Delta$, there is no node, or edge in G labeled d (our inference algorithm does not infer an embedding mechanism for these productions). ■

3.8 Definition of Data Structures Used in the Algorithm

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, …, 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, …, 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 in the algorithm for node graph grammar inference and no more than two vertices in edge grammar inference, copies of S_G.

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.

3.9 Node Replacement Graph Grammar in Hierarchy of Graph Grammars

We encountered in the existing literature a classification of graph grammars based on the embedding mechanism [Kim97]. 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 10 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 dissertation. The set of grammars inferred by Jonyer et al. (Jonyer, 2002, 2004) we call *chain grammars*. Chain grammars describe graphs or a 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*→ *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 shown later in Figure 28.

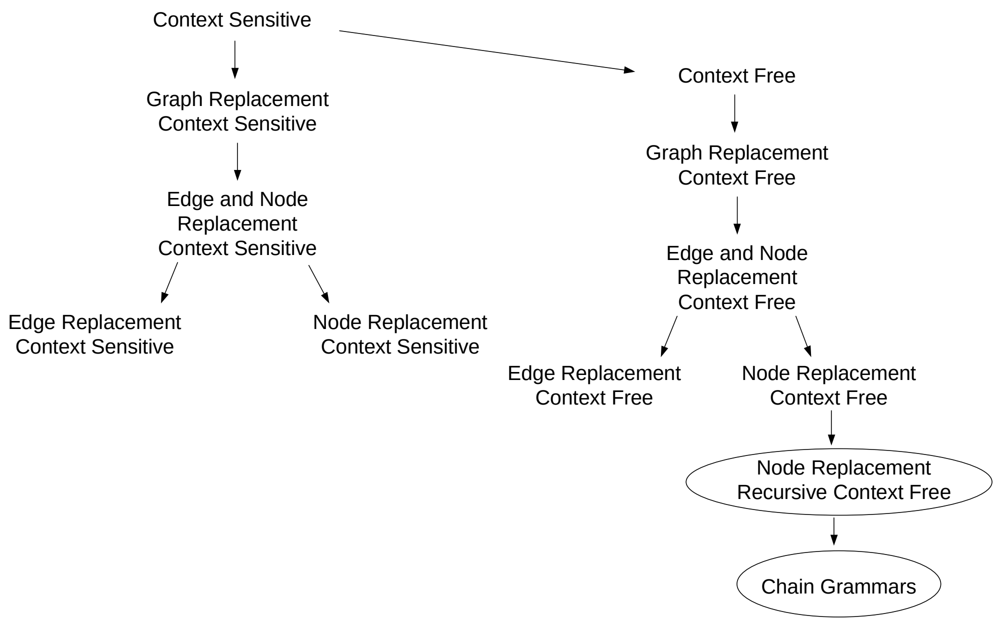

Context Sensitive
Context Free
Graph Replacement Context Sensitive
Graph Replacement Context Free
Edge and Node Replacement Context Sensitive
Edge and Node Replacement Context Free
Edge Replacement Context Sensitive
Node Replacement Context Sensitive
Edge Replacement Context Free
Node Replacement Context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

Figure 10: Hierarchy of graph grammars.

CHAPTER 4

THE GRAPH GRAMMAR INFERENCE ALGORITHMS

In this chapter we introduce two algorithms. The first algorithm is for node replacement graph grammar inference and the second for edge replacement graph grammar inference. We start from an informal description of the first algorithm using an example. Then we show pseudocode and explain how we detect overlap between substructures and determine connection instructions. Then we show how to find overlap by two nodes used to propose an edge replacement grammar. We define the virtual and real non-terminal edges in edge replacement graph grammars.

4.1 Node Replacement Graph Grammar Inference Algorithm

We will first describe the algorithm informally allowing for an intuitive understanding of the idea. The example in Figure 11 shows a graph composed of three overlapping substructures. The algorithm generates candidate substructures and evaluates them using the following measure of compression,

$$\frac{size(G)}{size(S)+size(G\,|\,S)}$$

where G is the input graph, S is a substructure and $G\,|\,S$ 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 (Cook 1994). Either of these measures can be used to guide the search and determine the best graph grammar.

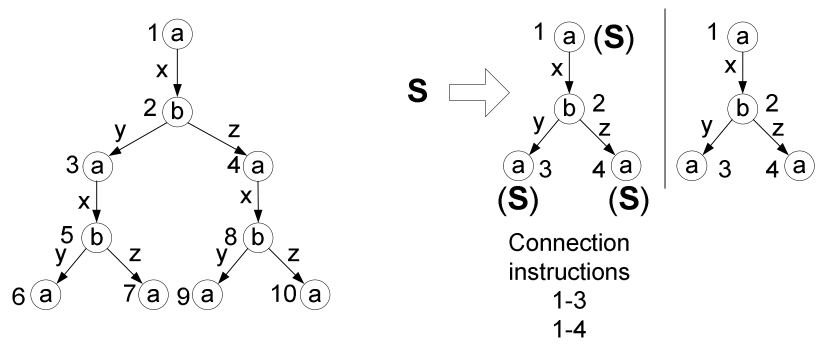


Figure 11: A graph with overlapping substructures and a graph grammar representation of it.

In Figure 11, the subgraphs overlap at nodes 3 and 4. The algorithm starts by finding nodes with the same label. There are seven nodes labeled “a” and three nodes labeled “b”. The single node labeled “a” becomes a candidate substructure with seven instances I_1={1}, I_2={3}, I_3={4}, I_4={6}, I_5={7}, I_6={9}, I_7={10}. The numbers in parentheses refer to the nodes in Figure 11. This initial substructure will be expanded by a node and an edge in each iteration of the algorithm’s main discovery loop. Similarly, the initial substructure of a node labeled “b” and its instances are determined. Both of these substructures are expanded simultaneously. Let us follow the expansion of only one substructure, which starts from all nodes labeled “b.” Table 1 gives the instance expansion at every step and a substructure value. We expand the instances **I** by edge labeled y and a vertex labeled a, which gives us the set of instances **I’.** Instances **I** can also be expanded by edge z or x. Similarly, we expand **I’** by edge z and a vertex a, which gives us **I''**. **I'** can also be expanded by edge x. We omit in Table 1 alternative expansions of **I** by z, x and **I'** by x. These additional expansions are part of our algorithm. They lead to the same solution. When the set of instances **I''** is expanded by the edge with label x, we detect an overlap, i.e., two or more instances share the same node. The overlapping instances of the substructure allow us to propose the recursive graph grammar shown on the right of Figure 11. This grammar can compress the entire graph to one node and has a better substructure value than any other substructure discovered so far.

.

Table 1: Expansion of instances which start from nodes labeled "b" in Figure 11.

Expansion	Instances	$\frac{size(G)}{size(S)+size(G\mid S)}$
initial instances	**I ={** I_1={2}, I_2={5}, I_3={8}**}**	19/(1+19)=0.95
I expanded by y	**I' ={** I_1={2, 3}, I_2={5, 6}, I_3={8, 9} **}**	19/(3+13)=1.19
I' expanded by z	**I''={** I_1={2, 3, 4}, I_2={5, 6, 7}, I_3={8, 9, 10}**}**	19/(5+7)=1.58
I'' Expanded by x	**I''' ={** I_1={2, 3, 4, 1}, I_2={5, 6, 7, 3}, I_3={8, 9, 10, 4}**}** (overlap !)	19/(7+1)=2.38

The grammar from Figure 11 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 11 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”.

Our graph grammar inference method is based on Cook et al.’s [Cook94] substructure discovery algorithm called Subdue. Subdue is looking for repetitive, highly-compressing subgraphs. The algorithm starts by finding all nodes with the same label. It maintains a list of the best subgraphs found so far. In each iteration new candidates for the best subgraphs are created by expanding all the subgraphs in the list by one edge or an edge and a node. Then, candidates for the best subgraphs are evaluated. In the evaluation process, every occurrence of a candidate subgraph within the entire graph is temporarily replaced by a new node. The compression achieved with this replacement is measured by calculating minimum description length or size (number of nodes + number of edges) of an original and compressed graph. Only subgraphs with the highest compression ratio remain in the list of the best subgraphs. Subdue spends a majority of computation time on isomorphism testing. Subdue has been applied to data from several domains, including DNA, chemical compounds, seismic events, aviation incident reports, and social networks [Su99, Chittimoori99, Gonzalez00, Mehta03].

Algorithm 1 is our grammar discovery pseudocode. The function INFER_GRAMMAR is similar to the descriptions of Cook et al. [Cook94] for substructure discovery and Jonyer et al. [Jonyer02] 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 12 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 12 is a continuation of the example in Figure 11. The numbers in parentheses refer to nodes of the graph in Figure 11.

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 [Cook94, Jonyer02, Jonyer04].

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 12 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 12, 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 12. Having the number of node $v \in G$ we find corresponding to v two node numbers of instance graphs $v_I \in I_1$ 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 12, 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.

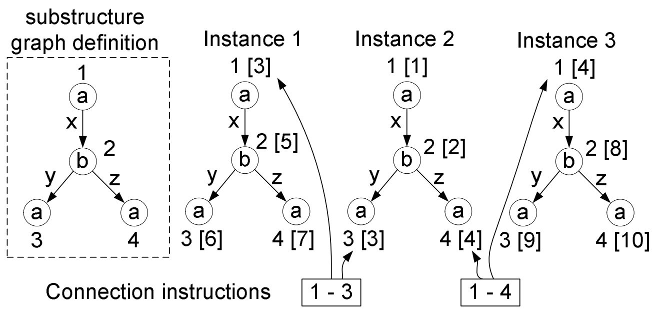


Figure 12: Substructure and its instances while determining connection instructions (continuation of the example from Figure 11).

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. Every recursive instance is compressed to a single node in the evaluation process.

Subdue uses a heuristic search whose complexity is polynomial in the size of the input graph [Cook00]. 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 $O(\mathrm{V}^2)$ time. The number of pairs of instances is no more than V^2, so the entire overlap test will not take more than $O(\mathrm{V}^4)$ time.

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 $S \in Q$
    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 $Limit \le 0$
    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.  $recursiveSub \rightarrow connectionInstructionList$ = { }
    2.  $recursiveSub \rightarrow Instances$ = { }
    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 node $v \in G$ )
    5.          $v_I$ = GET_INSTANCE_NODE($v$, $I_1$ )
    6.          $v_I'$ = GET_INSTANCE_NODE($v$, $I_2$ )
    7.          **if** ( $(v_I, v_I') \notin$ ($recursiveSub \rightarrow connectionInstructionList$) )
    8.              Add $(v_I, v_I')$ to ($recursiveSub \rightarrow connectionInstructionList$)
    9.          **end if**
    10.         **if** $I_1 \cap IR_k \neq \emptyset$ or $I_2 \cap IR_k \neq \emptyset$ , where $IR_k$ is any member of $recursiveSub$
        $\rightarrow Instances$
    11.             modify $IR_k$ ,    $IR_k = IR_k \cup I_1 \cup I_2$ **else**
    12.             create new entry $IR_k = I_1 \cup I_2$ and add it to $recursiveSub \rightarrow Instances$
    13.         **end if**
    14.     **end if**
    15. **end for**
    16. **return** $recursiveSub$
```

In our first example from Figure 11, 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 13 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 13 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 13 is the actual inferred grammar.

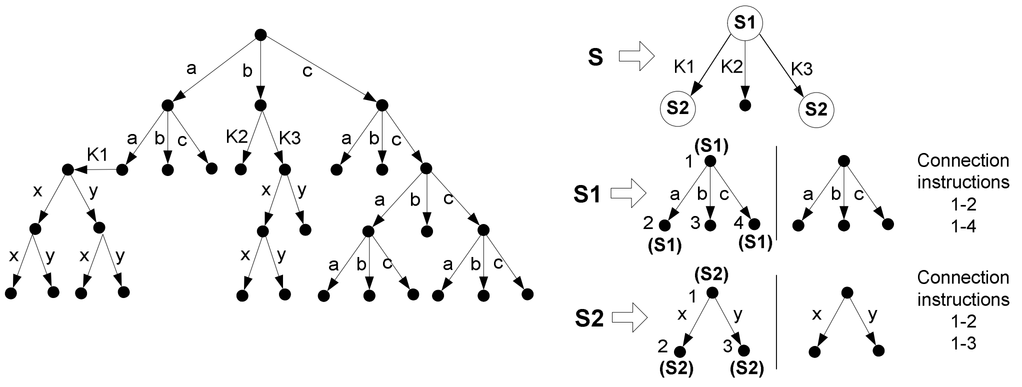


Figure 13: The tree (left) and inferred tree grammar (right).

We notice that in Figure 13 productions S1 and S2 are recursive with two connection instructions but production S is not recursive and does not have connection instructions. 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. 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 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 discuss possible solutions in the future work section of the dissertation.

4.2 Edge Replacement Graph Grammar Inference Algorithm

There is overlap in the recurring patterns or motifs representing the building blocks of networks in nature. Palla et al. [Palla05] point out the existence of an overlap between parts of graphs representing social networks and proteins. They call them overlapping communities. In our method of graph grammar inference we search for overlap between isomorphic subgraphs of a graph. The overlap allows for proposing recursive graph-grammar productions. The first approach was to search for overlap by a single node, which led to developing an algorithm for inference of Node Replacement Recursive Graph Grammars (Kukluk06). Now, we describe an extension to the node-replacement approach that allows inference of Edge Replacement Recursive Graph Grammars. One limitation of node replacement grammars is shown in Figure 14, where all the nodes have the same labels. We infer a node replacement grammar from the graph in Figure 14 (a). The grammar inferred in Figure 14 (b), if used for generation, would replace node 2 with node 1 of an isomorphic copy of an instance. This grammar can generate the graph in Figure 14 (c), but it cannot regenerate the original graph in Figure 14 (a). Motivated by this limitation of node replacement grammars, we extended the approach to edge replacement grammars by allowing for overlap between two nodes. We discuss inference error and how different search-guiding measures influence error. We also address how different numbers of labels used in the graph affect the inference error.

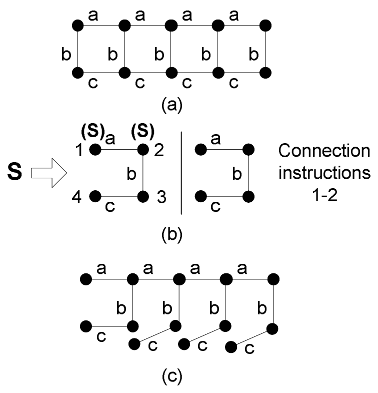


Figure 14: Graph with overlapping squares (a), inferred node-replacement grammar (b), and graph generated from inferred grammar (c)

In the definition of edge replacement recursive graph grammars we have two types of productions. The *recursive production* has a graph in its definition with one or more non-terminal edges which we can replace with the structure of the graph to which these non-terminal edges belong. 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. *Non-recursive productions* replace a single node with a graph. Our algorithm does not infer the embedding mechanism for non-recursive productions. In every iteration of the grammar inference algorithm we are finding only one production, and it is ether *non-recursive* or *recursive*. The reader can refer to examples in Figure 15 and Figure 33 while examining the definition. In Figure 15 (a) we see an example of the grammar used for generation and in Figure 15 (b) the equivalent inferred grammar.

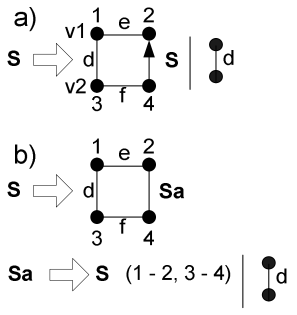


Figure 15: The original grammar (a) used to generate examples and the inferred grammar (b).

The edge replacement algorithm operates on a data structure called a *substructure* (similar to the algorithm for node replacement grammar inference) which in Algorithm 2 we represent by *S*. A substructure consists of a graph definition of the repetitive subgraph and its instances. We illustrate it in Figure 16. We defined two functions in Algorithm 2: INFER_GRAMMAR and RECURSIFY_SUBSTRUCTURE. The first function is consistent with Cook et al.'s [Cook 1994] algorithm. Initially, the graph definitions of substructures are single nodes, and there are as many substructure 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 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. For more details about the algorithm see [Cook94, Jonyer02, Jonyer04].

Our addition to Cook et al.'s [Cook94] algorithm is the procedure RECURSIFY_SUBSTRUCTURE. This procedure takes substructure *S* and examines its instances for overlap at line 4. If two nodes v_1, v_2 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, 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. In our experiments we used two evaluation measures:

$$\frac{size(G)}{size(S) + NT + [size(G \mid S)]^2},$$

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

Algorithm 2 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 $S \in Q$
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 $Limit \le 0$
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
           $recS \rightarrow Instances$
14. return recS
```

Let us call these measures a *measure with square* (first) and a *measure without square* (second). Initially, we only used measure without square but with this measure the inferred graph is often missing one edge in comparison to the original grammar. Therefore, we introduced the measure with square which includes the missing edge to the inferred grammar. *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 reason of using measure with square is to put more emphasis on compression than it is in the case of the measure without square. This emphasis of compression can be achieved in two different ways: (1) minimizing value of *size(S)* or (2) enlarging value of *size(G|S)*. We chose to enlarge *size(G|S)* by applying exponent 2, but it is not a strong preference and other exponents or algebraic methods of emphasizing the *size(G|S)* in reference to *size(S)* are expected to have similar effects. The *size* we measure with two methods: (1) Minimum Description Length (MDL) and (2) number of nodes plus number of edges. We discuss the effects of these different evaluation methods on inference error later in the section dedicated to experiments. The original Subdue algorithm uses a heuristic search whose complexity is polynomial in the size of the input graph. Our additions have their main computations in checking for overlap between instances of substructures, and they do not change the complexity of this algorithm.

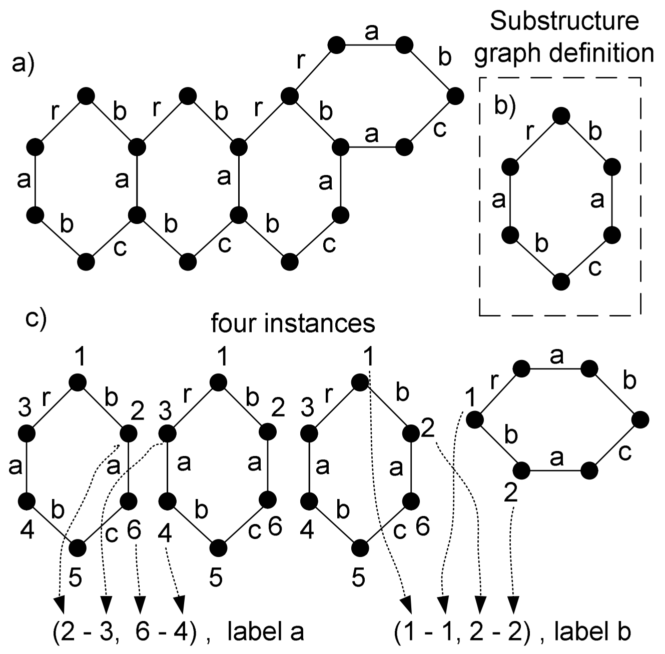


Figure 16: The input graph (a), substructure graph definition (b) and four overlapping instances of repetitive subgraph (c).

The algorithm can learn grammars with multiple productions. When we find a production (recursive or not), we compress a portion of the graph described by the production in line 19 of the algorithm. Every connected subgraph described by the production is compressed into a node. Then we perform inference again on the compressed graph. We progress with alternating inference and compression until we cannot compress the graph any more.

CHAPTER 5

GENERATION OF GRAPHS FROM STOCHASTIC GRAPH GRAMMARS

In this chapter we describe the graph generator. We use this generator to generate graphs in experiments reported in the following chapters. We explain how we add noise and corruption to the generated graphs.

5.1 Overview

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 or edge by a graph until all nodes and edges are terminal. We identify non-terminal nodes and edges by labels S#, where # is an integer. We assign a real number from 0% to 100% to every production. This number is the probability of selecting the production. We mark two nodes of a graph used in a production with v1 and v2 to indicate connection nodes to the graph with non-terminal edges.

Let us examine the example from Figure 17, where we show the input file for the example grammar, its graphical representation and one, possible, derivation (generation) of a graph. We have a graph grammar with three productions. We see the format of an input file where every production starts from S# and is following by a probability and a graph definition. A node definition begins with the letter v followed by the node number. One node is marked v1 and one v2. An edge definition begins with the letter e and is following by the first node number, the second node number, and the edge label. The first production has assigned probability 100% and is the starting graph of a grammar. It has one non-terminal node labeled S1. S1 can be replaced with two graphs: a square with probability 60% or two edge graph with probability 40%. We additionally mark node v1 with a '*'' as we use '*' in other figures in this work where we limit productions to node replacement productions. In the example of the derivation in Figure 17 we show the steps to generate the graph assuming that the generator selected in order the first, second and third productions.

5.2 Graph Grammar Generation Algorithm

Algorithm 3 describes the generation process. There are five input parameters to the function GENERATE_GRAPH. 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* or the number of generated graphs is larger than *maxIterations.* We distinguish two different distorting operations to the graph generated from 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. We use two functions of Algorithm 3 to perform these described distortions: CORRUPT_GRAPH_STRUCTURE and ADD_NOISE_TO_GRAPH.

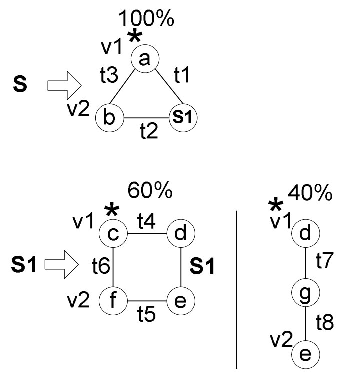
The generation process starts by selecting a starting graph. There are one or more productions whose left side is S. The starting graph is selected from the right side of these productions. Then the starting graph is examined for non-terminal edges and nodes. The function GENERATE_GRAPH_FROM_GRAMMAR of Algorithm 3 operates on the starting graph. When a non-terminal node or edge is identified, we call the recursive function EXPAND_GRAPH_BY_EDGE_OR_NODE (Algorithm 4). This function identifies the graph to be added from the right side of production rules whose left side matches the label of a non-terminal node or edge. After replacing the non-terminal edge or node with a graph, the procedure searches through the newly added nodes and edges for non-terminals. If it finds one, it calls itself recursively. The process continues until the generated graph does not have any non-terminals.

Input file

```
S
100%
v1 1 a
v2 2 b
v 3 S1
e 1 3 t1
e 3 2 t2
e 2 1 t3

S1              S1
60%             40%
v1 1 c           v1 1 d
v2 2 f          v2 2  e
v 3 d           v 3 g
v 4 e           e 1 3 t7
e 1 3 t4        e 3 2 t8
e 3 4 S1
e 4 2 t5
e 2 1 t6
```

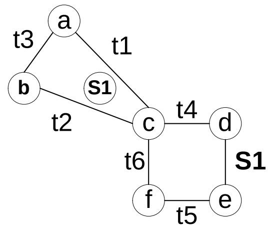
Graphical representation



Example derivation

Let as assume that generator selected in order first, second and third production. The derivation of a graph will be:

1) First production is applied and starting graph is a triangle with one non-terminal node S1.
2) Second production is applied and all incident edges to node S1 are redirected to node labeled c



3) Node S1 is deleted
4) Third production is applied and edge S1 is replaced by graph from right side of third production

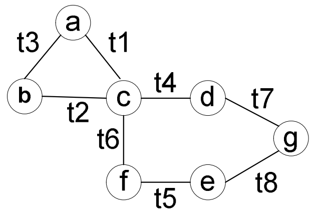


Figure 17: Input format to graph generator and example derivation.

Algorithm 3 Graph generation from graph grammar

```
GENERATE GRAPH (grammar, minNodes, maxNodes, noise, maxIterations)
 1. while ( NOT( $minNodes \leq$ numberNodes(graph) $\leq maxNodes$ ) AND
 2. (number of generated graphs $\leq maxIterations$)) do
 3.     graph=GENERATE_GRAPH_FROM_GRAMMAR(grammar)
 4.     CORRUPT_GRAPH_STRUCTURE (graph, noise)
 5.     ADD_NOISE_TO_GRAPH (graph, noise)
 6. end while
 7. return graph

CORRUPT_GRAPH_STRUCTURE (graph, noise)
 1. while (number of redirected edges $\leq$ noise*number_of_edges(graph)) do
 2.     select random edge $e = (v_1, v_2)$
 3.     replace nodes $v_1, v_2$ with new nodes $v'_1, v'_2$ randomly selected from the graph
 4. end while
 5. return graph

ADD_NOISE_TO_GRAPH (graph, noise)
 1. while ( number of added nodes $\leq$ (noise/(1- noise))*number_of_nodes
    (graph) do
 2.     add new node to the graph with a label randomly selected from labels used
    already in the graph
 3. while ( number of added edges $\leq$ (noise/(1- noise))*number_of_edges
    (graph) ) do
 4.     select two nodes $v_1, v_2$ from existing graph and add new edge $e = (v_1, v_2)$ with
    label
        already used in the graph
 5. return graph

GENERATE_GRAPH_FROM_GRAMMAR (grammar)
 1. Select starting graph from right side of productions whose left side is S. Selection
    is made according to probability distribution determined by probabilities given in
    input file for every production.
 2. for all edges $e \in graph$
 3.     if $e$ has non-terminal label
 4.         node_or_edge=edge
 5.         EXPAND_GRAPH_BY_EDGE_OR_NODE (graph, NULL, e,
    node_or_edge)
 6.     end if
 7. for all nodes $v \in graph$
 8.     if $v$ has non-terminal label
 9.         node_or_edge=node
10.         EXPAND_GRAPH_BY_EDGE_OR_NODE (graph, v, NULL,
    node_or_edge)
11.     end if
12. return graph
```

Algorithm 4 Recursive expansion of a graph by an edge or node.

```
EXPAND_GRAPH_BY_EDGE_OR_NODE (graph, v, e, node_or_edge)
 1. From all productions whose left side is the same as label of v (or e if
    node_or_edge=edge) select graphToAdd from right side of the
    production according to probability distribution determined by
    probabilities given in input file for every production.
 2. Add all edges and nodes of graphToAdd to graph
 3. if  node_or_edge==node
 4.    Identify a connectingNode of graphToAdd (this node is marked in
    the input file)
 5.    Make all edges incident to $v \in graph$ incident to connectingNode
 6.    Delete v
 7. end if
 8. if  node_or_node==edge
 9.    Identify a connectingNode1 and connectingNode2 of graphToAdd
    (marked in the input file)
10.    Given input edge $e=(v_1,v_2)$, $e \in graph$, make all edges incident to $v_1$
    incident to connectingNode1
       and all edges incident to $v_2$ incident to connectingNode2
11. Delete $e, v_1, v_2$
12. end if
13. for all edges $e \in graphToAdd$
14.    if  e has non-terminal label
15.       node_or_edge=edge
16.       EXPAND_GRAPH_BY_EDGE_OR_NODE (graph, NULL, e,
    node_or_edge)
17.    end if
18. for all nodes $v \in graphToAdd$
19.    if  v has non-terminal label
20.       node_or_edge=node
21.       EXPAND_GRAPH_BY_EDGE_OR_NODE (graph, v, NULL,
    node_or_edge)
22.    end if
23. return graph
```

CHAPTER 6

EXPERIMENTS WITH NODE REPLACEMENT GRAPH GRAMMARS

In this chapter we present experiments to analyze the performance of the node replacement graph grammar inference algorithm. We begin the chapter with methodology and the MDL role in determining complexity of a grammar. Then we introduce the definition of an error. Next, follow several experiments showing how error depends on noise, complexity of a grammar, number of labels, and size of a graph. We examine experiments indicating limitations of the algorithm. The last experiment shows the graph grammar inferred from chemical structure. The chapter ends with conclusions.

6.1 Methodology

Having our algorithm 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 18 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 18. 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 algorithm does not infer probabilities. Oates et al. [Oates03] 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.

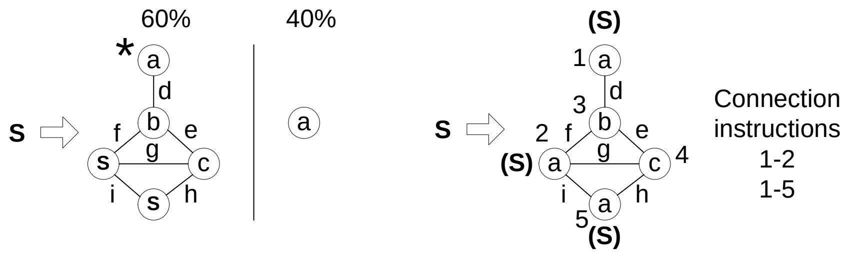


Figure 18: An example of a graph grammar used in the experiments.

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 [Read98]. We show them in Figure 22. 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 algorithm 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.

6.2 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 [Cook94] 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.

Next, we give an example of calculating the MDL of a graph using the graph structure from Figure 18. The adjacency matrix of the graph and the MDL calculation are as follows.

$$\begin{array}{c|ccccc} & a & a & b & c & a \\ a & & & 1 & & \\ a & & & 1 & 1 & 1 \\ b & & & & 1 & \\ c & & & & & 1 \\ a & & & & & \end{array}$$

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

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

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

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

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

6.3 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 [Bunke1983]. For more details see also [Cook94].

In our experiments we measure an error based on structural difference. Another approach to measuring the accuracy of the inferred grammar would be based on a graph grammar parser. We would consider accurate the inferred grammars that can parse the input graph. Graph grammar parser would require subgraph isomorphism test which is computationally expensive and much more difficult in implementation than the error measure we are using. For these reasons we did not pursue implementation of graph grammar parser.

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.

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

$\text{matchCost}(g_1, g_2)$ 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

In Figure 19 we see two productions we use in our example of error calculation. The production on the left is the original production and the production on the right is inferred production. The production on the left has two non-terminals, $\#NT = 2$. Production on the right has one inferred connection instruction, $\#CI = 1$. There are three nodes and two edges in the graph structure on the right, $\text{size}(g_1) = 5$. We can transform graph from the left to the graph on the right in Figure 19 by removing two edges and one node from the graph on the right. Therefore $\text{matchCost}(g_1, g_2) = 3$ The inference error

$$Error = \min\left(1, \frac{\text{matchCost}(g_1, g_2) + |\#CI - \#NT|}{\text{size}(g_1) + \#NT}\right) = \min\left(1, \frac{3 + |1-2|}{5+2}\right) = 0.57$$

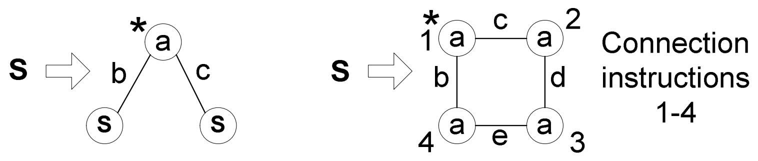


Figure 19: Graph grammar inference error.

6.4 Experiment 1: Error as a Function of Noise and Complexity of a Grammar

We used twenty nine graphs from Figure 22 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 20**.** 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 20. 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 21 has the same results as Figure 20 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.

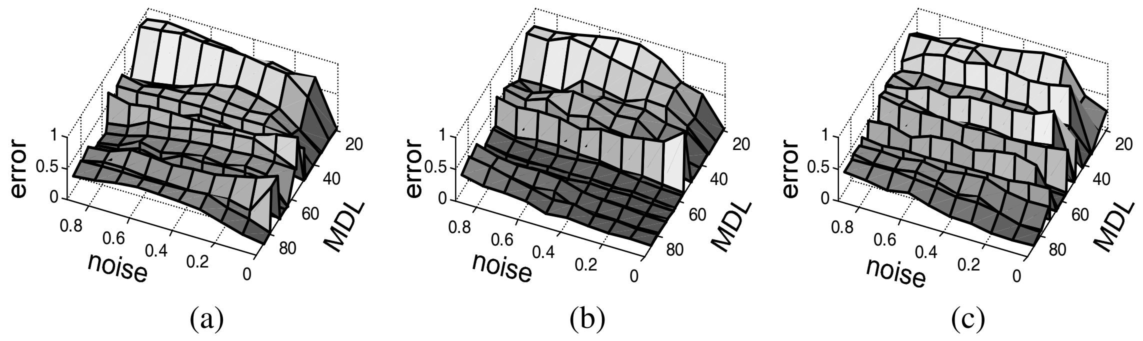
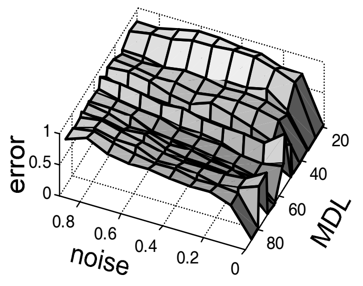
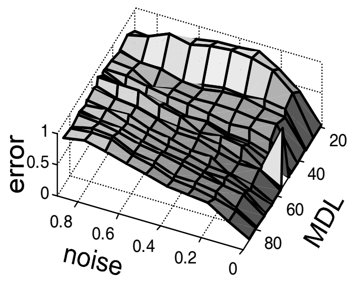


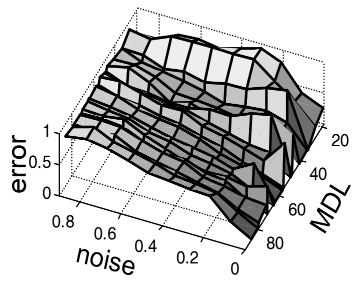
Figure 20: Error as a function of noise and MDL where graph structure was not corrupted: one non-terminal (a), two non-terminals (b), and three non-terminals (c)



(a)



(b)



(c)

Figure 21: Error as a function of noise and MDL where graph structure was corrupted: one non-terminal (a), two non-terminals (b), and three non-terminals (c).

We see trends in the plots in Figure 20 and Figure 21. 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 20).

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 22 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. 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 22. 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.

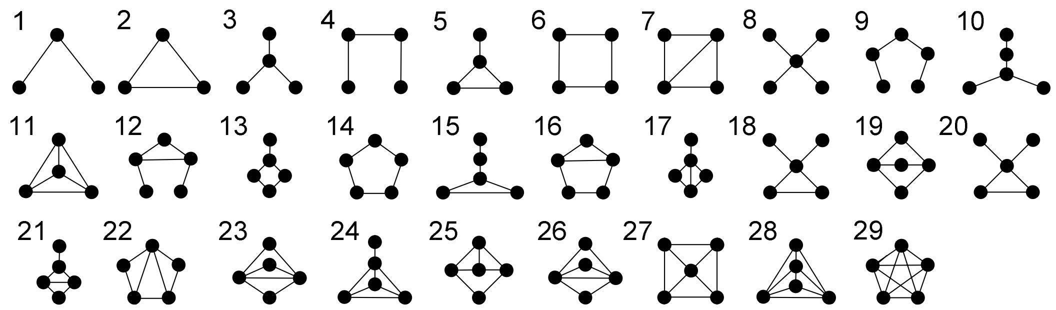


Figure 22: Twenty nine simple connected graphs ordered according to non-decreasing MDL value.

Table 2: Twenty nine simple graphs ordered according to increasing average inference error of six experiments in Figure 20 and Figure 21. The numbers in the table refer to structures in Figure 22.

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		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		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		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		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 23: An inference error where larger graph structure is proposed: original grammar (a) and inferred grammar (b). The structure has significant error, yet does subjectively capture the recursive structure of the original grammar.

6.5 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 25. 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.

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 24 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.

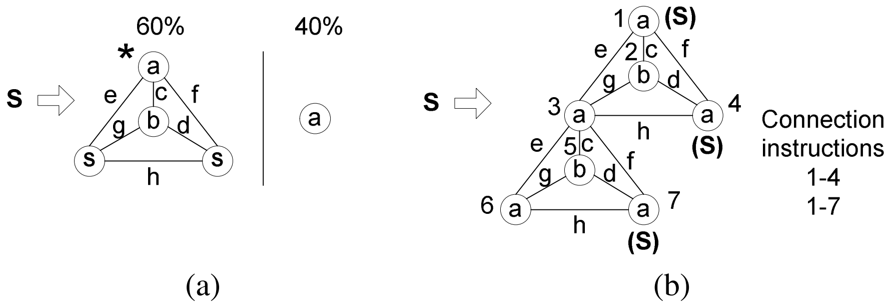


Figure 23: An inference error where larger graph structure is proposed: original grammar (a) and inferred grammar (b).

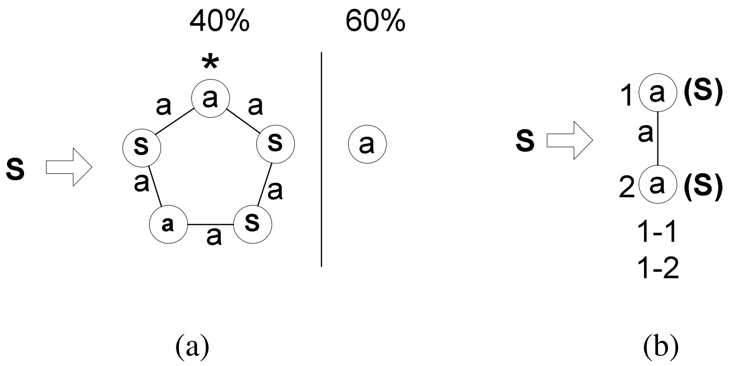


Figure 24: Error where inferred grammar is reduced to production with single edge: original grammar (a) and inferred grammar (b).

6.6 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 26 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.

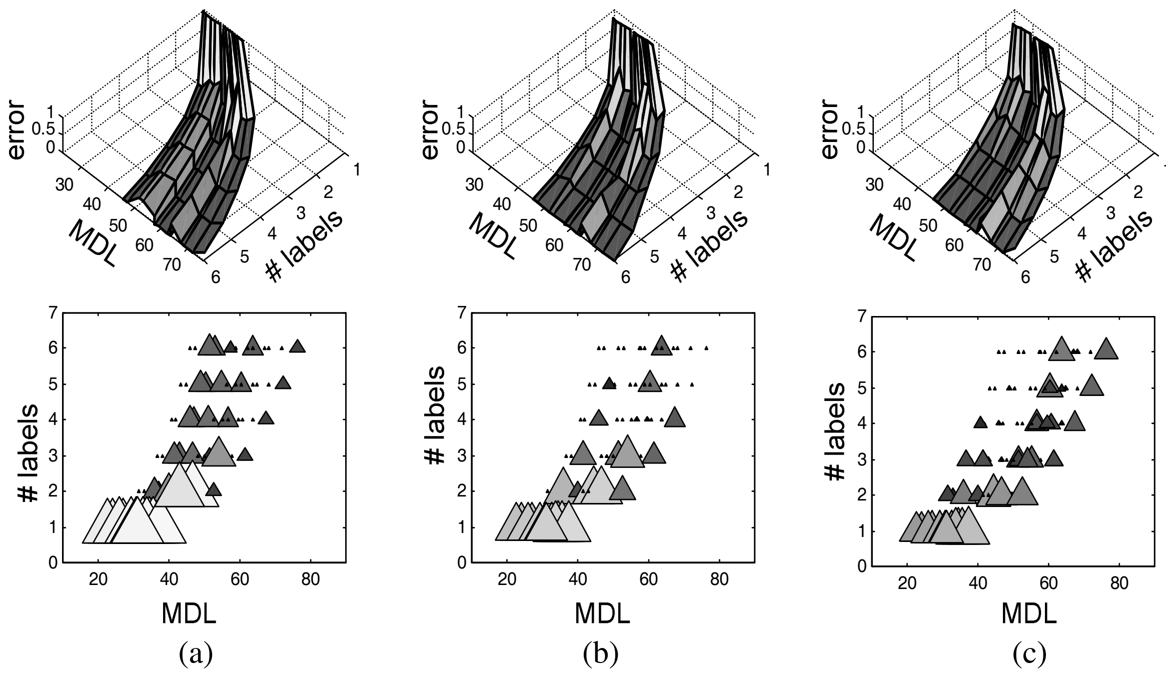
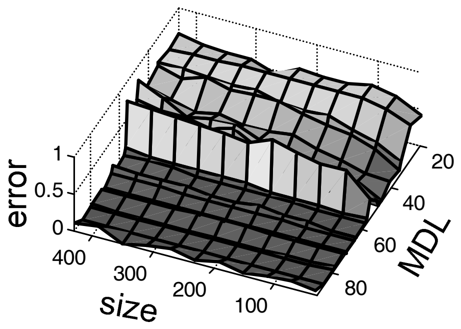
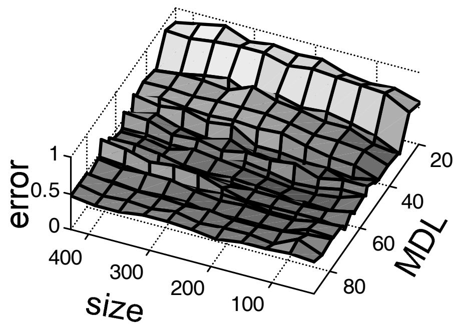


Figure 25 : Error as a function of MDL and number of different labels used in a grammar definition: one non-terminal (a), two non-terminals (b), and three non-terminals (c). .



(a)



(b)

Figure 26: 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

6.7 Experiment 4: Limitations

In Figure 27 we show an example illustrating the limits of our approach. In Figure 27 (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 27 (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.

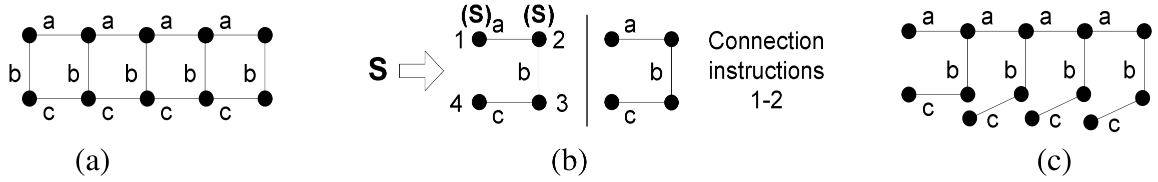


Figure 27: Graph with overlapping squares (a), inferred grammar (b), and graph generated from inferred grammar (c)

Figure 28 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 28. 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 multiple productions in one iteration or allow for multiple compression passes on the input graph.

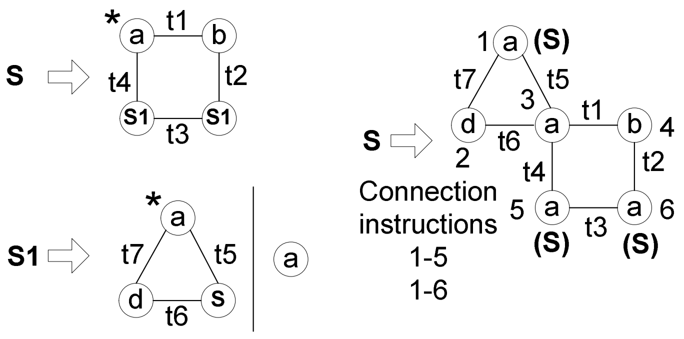


Figure 28: The grammar with alternating productions (left) and inferred grammar (right).

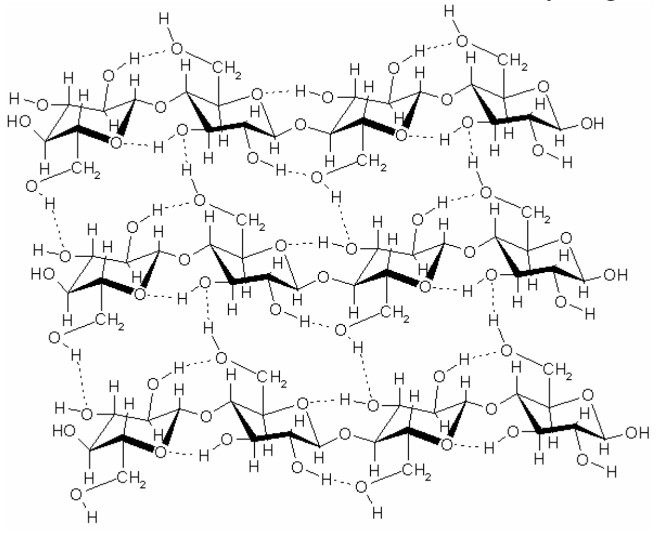
6.8 Experiment 5: Chemical Structures

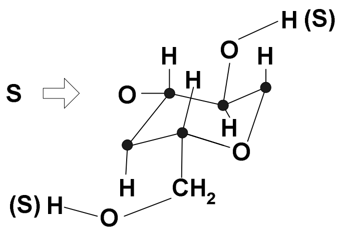
As an example from the real-world domain of chemistry, we use four chemical structures as the input graphs in our next experiment. Figure 29 and Figure 30 show the structures of the molecules and the grammar productions we found in these structures. The first structure in Figure 29 is the structure of cellulose with hydrogen bonding. The second molecule is macrocyclic gallium carboxylate [Uhl04]. 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 29 is water-soluble tin-based metallodendrimer [Schumann03]. 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 compresses into a single vertex. The graph on the right hand side of production S is a graph of a chemical structure compressed with S1.

In Figure 30 we have the structure of a dendronized polymer [Zhang03]. 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 30 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.

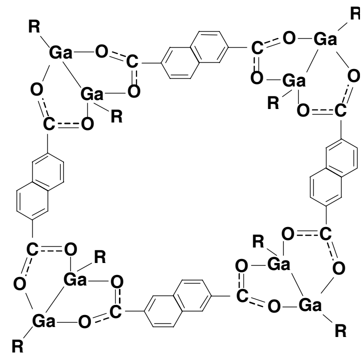
Chemical structure Inferred productions

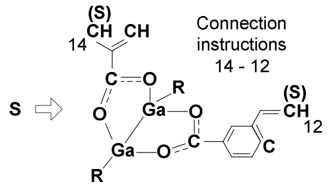
cellulose with hydrogen bonding





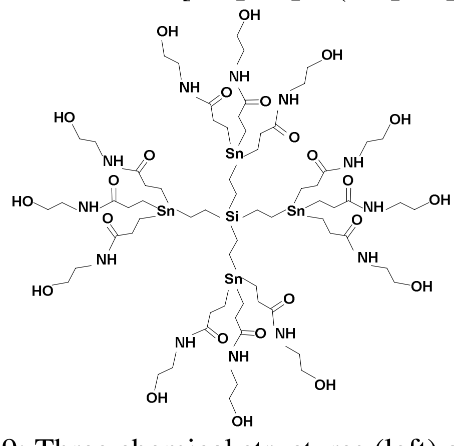
Macrocyclic gallium carboxylate





water-soluble tin-based metallodendrimer

$Si[CH_2CH_2Sn(CH_2CH_2CONHCH_2CH_2OH)_3)]_4$



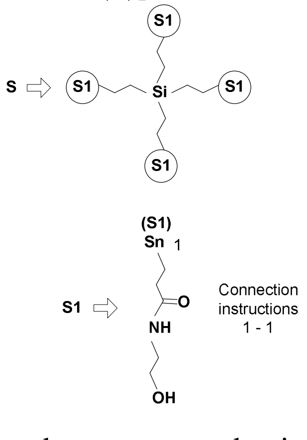
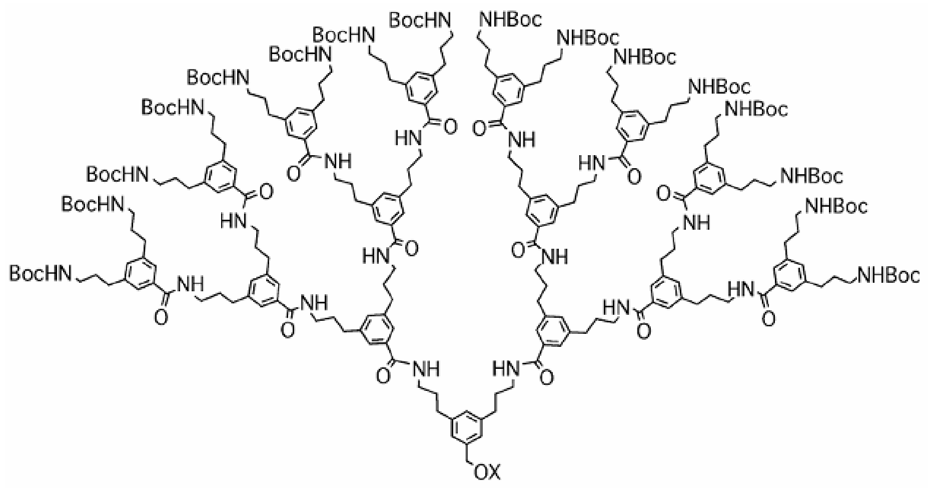


Figure 29: Three chemical structures (left) and the inferred grammar production (right).



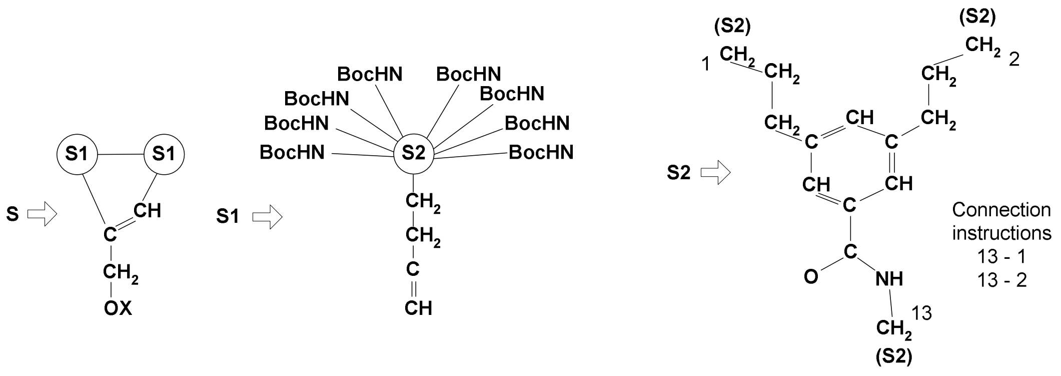


Figure 30: The structure of dendronized polymer and its representation in hierarchical graph grammar productions.

6.9 Experiment 6: Learning Curves

We wanted to examine the learning process on a graph grammar with several productions. Since there are an infinite number of different graph grammars, we decided to select one example with several different graph structures used in the grammar productions. We show this example in Figure 31, where we see the graph grammar used to generate graphs. There are five productions. The last production with only one node is a terminating production. Each graph in the first four productions had two non-terminal nodes. The first four productions are chosen with probability 0.1 in the generation process. The terminating production is chosen with probability 0.6.

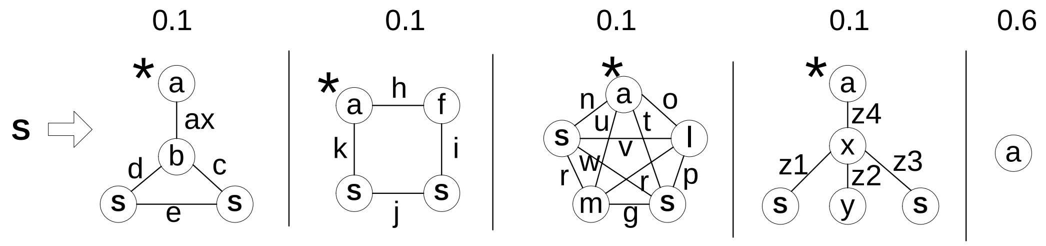


Figure 31: Graph grammar used for graph generation

We generate sets of graphs with 10, 20, 30, and up to 100 graphs generated from the grammar in Figure 31. Every graph in the set has 30 to 40 nodes. We compare the first four grammar productions found by our algorithm to the original grammar in Figure 31. As a measure of an error, we use the minimal match cost of a transformation from one graph structure to the other, as described in section 8.3 where we talk about the measure of the error. We calculate the match cost of the structure of the graph from the first inferred grammar production to the four structures of the original productions and choose the smallest value. Then, we calculate the match cost of the structure from the second inferred production to the three structures from the original grammar not selected before and select the smallest value. Similarly, we find the smallest match cost between the structure of the third inferred production and the two structures left. The last inferred production we compare to the remaining production from the original grammar. The process of matching productions from the inferred grammar to the original grammar is greedy. It may not find the minimal error of matching four productions, because we do not explore all possible matchings. The inference error we compute as a sum of the four errors we just explained. We repeat generation and error determination thirty times and compute the average value of the error. In Figure 32 we show the grammar inference error and time as a function of the number of graphs in the input set. We see that time in the range 10 to 100 graphs has close to linear increase. The error decreases sharply as we increase the set of graphs from 10 to 30. The error does not reach zero. The input graph has now four patterns. We often infer productions which contain two of the patterns or a portion of two patterns which causes the error.

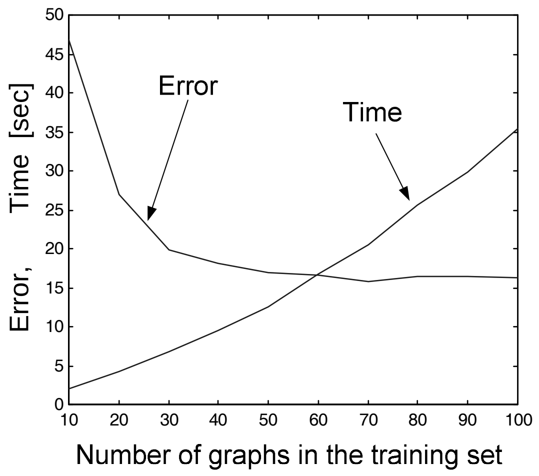


Figure 32: Error and time as a function of number of graphs in the training set.

6.10 Summary of Results and Conclusions

We described experiments with node replacement graph grammars. 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.

Grammars inferred by the approach developed by Jonyer et al. [Jonyer04] 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 27, 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 is a modification which allows for overlap larger than a single node, which we accomplish in the algorithm for Edge Replacement Recursive Graph Grammars. The next chapter presents experimental results using this algorithm.

CHAPTER 7

EXPERIMENTS WITH EDGE REPLACEMENT GRAPH GRAMMARS

In this chapter we describe experiments on edge replacement graph grammars. We show examples of productions inferred with real and virtual edges. We show how different evaluation measures, different graph structures, and noise, influence the inference error. Next, we report experiments with chemical structures. A summary of experiments and conclusion ends the chapter.

7.1 Introduction

We describe experiments on the algorithm for inference of edge replacement graph grammars. This method generates candidate recursive graph grammar productions based on finding 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 using the Minimum Description Length (MDL) of a graph and the size of a graph. 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.

7.2. 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:

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

$\text{matchCost}(g_1, g_2)$ 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

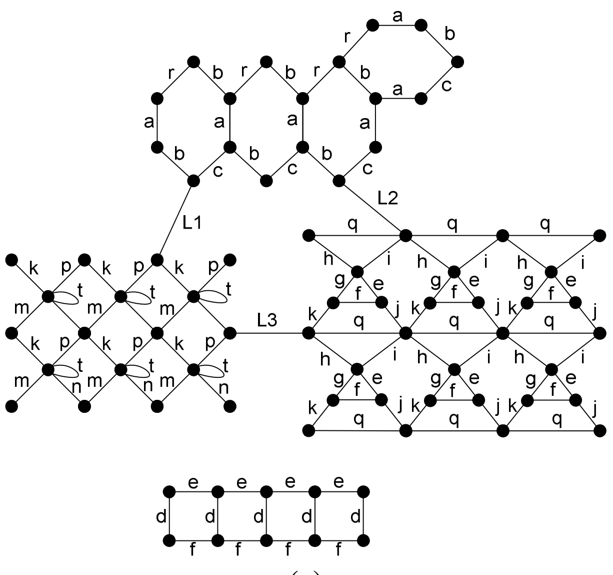
$\text{size}(g_1)$ is the sum of the number of nodes and edges in the graph used in the grammar production

$\text{matchCost}(g_1, g_2)$ measures the structural difference between two graphs with an algorithm for inexact graph match initially proposed by Bunke and Allermann [Bunke83]. For more details see also [Cook94]. 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 algorithm.

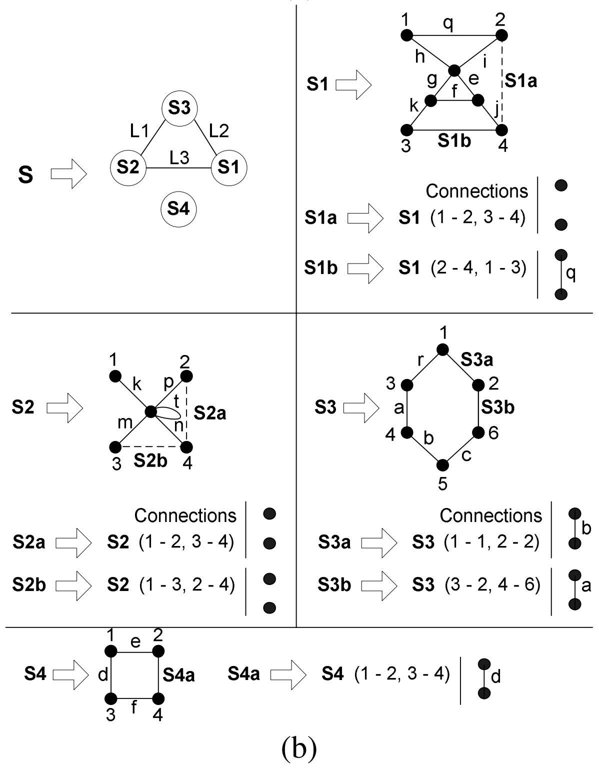
7.3. Experiment 1: Virtual and Real Edges in Productions

In Figure 33 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 verity of productions and the nature of edge replacement grammars our approach can handle. The input graph has four different repetitive patterns. We did not generate this graph. We constructed it manually such that we can find different productions in it. 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 for which we do not infer connection instructions. 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.

Productions S3 and S4 can regenerate the portions of the graph they describe. We consider the inferred grammar correct, although productions S1 and S2 cannot regenerate the structures from which we inferred them. In the generation process we replace a non-terminal edge with a graph and then if the expanded portion of a graph contains a non-terminal edge it is expanded further. In this paradigm there is no way that expanded portions of the generated graph will have any additional links or connections between them except the connection which includes two nodes of the expanded non-terminal edge. In order to regenerate structures covered by productions S1 and S2, we would need a more sophisticated generation mechanism with context sensitive embedding mechanism. This mechanism, inferred during induction, would indicate nodes to merge during the generation process.



(a)



(b)

Figure 33. The graph (a) and inferred grammar from this graph (b).

7.4. Experiment 2: Inference Error with Different Evaluation Measures

In Figure 34 we examine how inference error is affected by different evaluation measures. We see four plots. Each plot has seven points. Every point we found by generating a graph using the 9-cycle grammar shown in Figure 34. We assigned label “a” to six nodes and one edge of the cycle. All other nodes and edges have distinct labels which we omit in the figure. We generated thirty graphs with 40 to 60 nodes from the cycle grammar. We inferred the grammar from the generated graph and measured the inference error. We computed the average over thirty errors for the value of a single point in the plots. We examined the error value where we used MDL to measure size(G), size(S) and size(G|S) in two cases: (1) measure without square and (2) measure with square. We examined the same two cases while measuring size by adding the number of nodes and edges $size(g) = nodes(g) + edges(g)$.

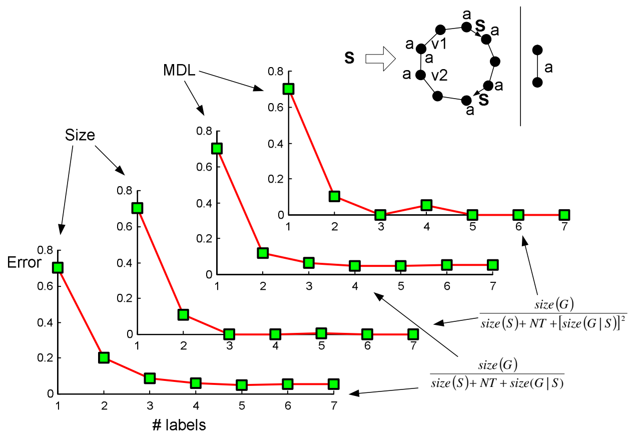
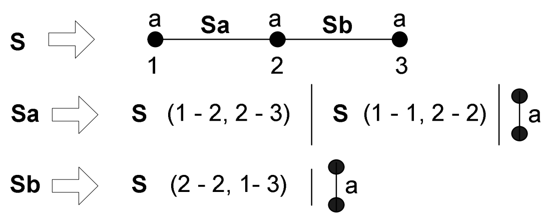
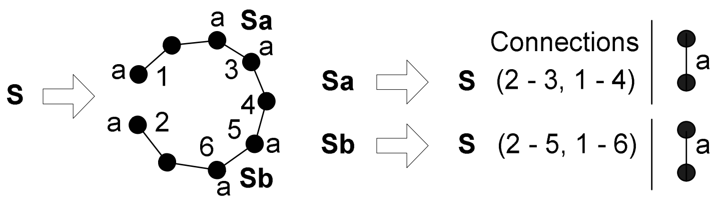


Figure 34: The influence on the inference error of evaluation measures using a graph grammar of a 9-cycle.

The error is high in all four cases when we use only one unique label in the graph. In Figure 35 (a) we show the grammar inferred from the graph with one label only. It is a two edge graph where both edges are non-terminals. This grammar can compress the entire graph and has a small structure and therefore scores high by our evaluation measure. The error drops to zero when we use three distinct labels in the graph with the square evaluation measure but it does not reach zero at all when we use the evaluation measure without the square. In Figure 35 (b) we show the grammar inferred using the evaluation measure without the square. We expect to infer the cycle but instead we infer the structure still missing one edge. The missing edge is the overlapping edge. This grammar with missing edge, however, compresses the input graph very well and therefore scores well. It leaves only one edge uncompressed from the entire graph. When we use $[size(G|S)]^2$ instead of $size(G|S)$ in our evaluation measure, compressing the entire graph, including the remaining single edge, becomes more important than small structure in the inferred grammar and we infer the complete cycle.



(a)



(b)

Figure 35: Two-edge grammar inferred from the graphs with only one label on nodes and edges (a) and inferred grammar with evaluation measure size(G)/(size(S)+NT+size(G|S)) where cycle is expected (b).

7.5. 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 36. Every point in the plots in Figure 36 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 36 and Figure 37 was assigned a label chosen from *k* distinct labels, where *k* is an integer from 1 to 7 in Figure 36 and from 1 to 16 in Figure 37. 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 the number of nodes, the error increases.

In Figure 36 increased structural complexity increases the inference error. In our experiments with node replacement graph grammar we noticed an opposite trend. With increased complexity measured by MDL we observe decreased error. These results suggest that identifying the non-terminal node and the pattern can be easier when structure complexity (ration of edges to nodes) increases. However, when we infer edge replacement graph grammars, identifying the non-terminal edge and the pattern is more difficult with increased structural complexity.

The highest error we had with complete graph. We show this case separately in Figure 37. 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 37, 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.

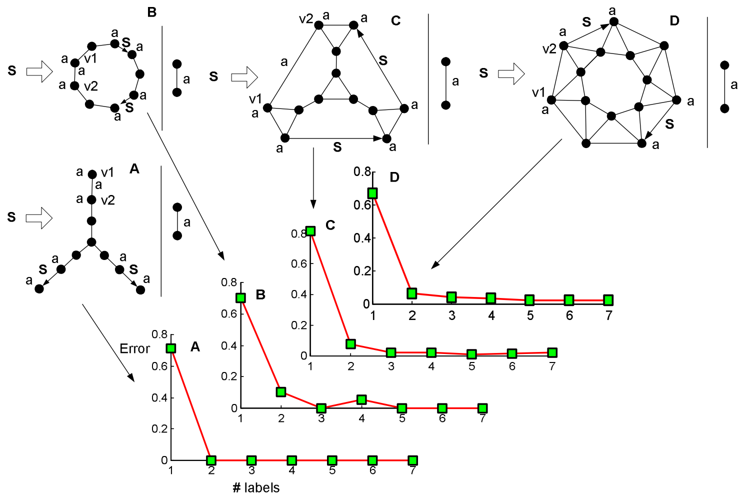
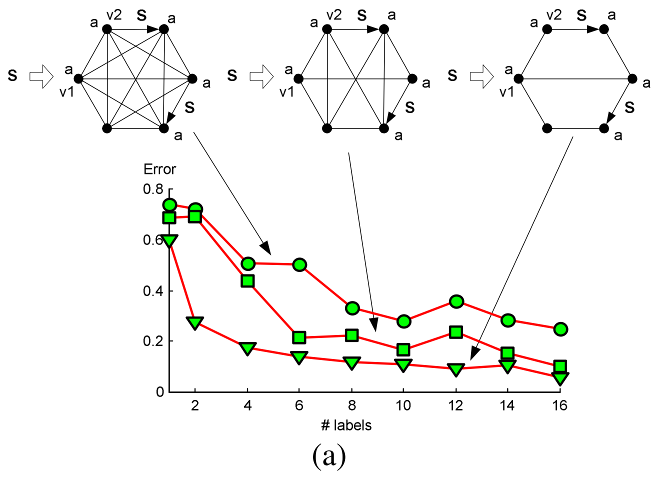


Figure 36: The influence on the error of different graph structures used in grammar productions.



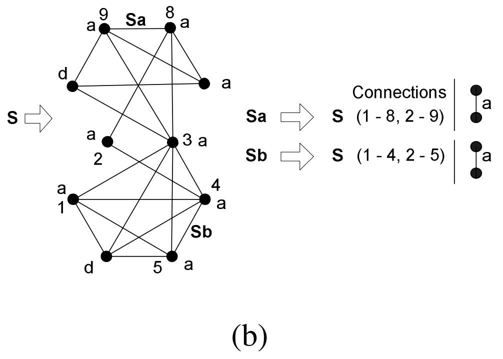


Figure 37: The change in the error with reduced number of edges from the complete graph structure (a) and an example of the inferred grammar (b).

7.6. Experiment 4: Inference Error in the Presence of Noise

In Figure 38 we show the results of an experiment where we generated graphs with the number of nodes from 40 to 60. The Peterson graph (Figure 38 (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 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 38 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.

7.7. Experiment 5: Chemical Structure

In Figure 39 (a) we show the chemical structure of G tetrad (Neidle, 1999). Versions of this structure are used in research on the HIV-1 virus (Phan, 2005). We converted this structure to a graph which we use as an input to our grammar inference algorithm. We found the grammar which represents the repetitive pattern of this chemical structure. We show the grammar in Figure 39 (b). This experiment demonstrates the potential application of our approach and also a weakness for further study. Although the grammar production we found captures the underlying motifs of the chemical structure, it cannot regenerate the original structure which has the ring form.

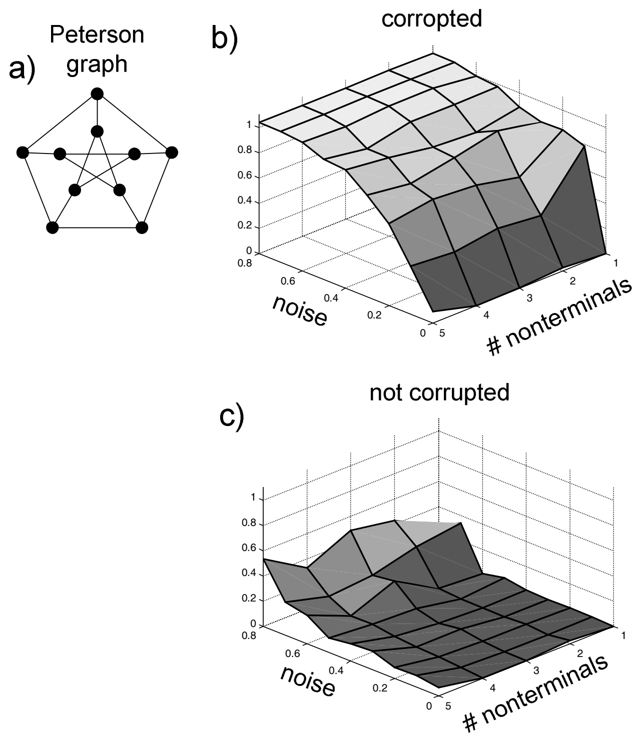


Figure 38: Inference error of a graph grammar with the Peterson graph structure in the presence of noise and different number of non-terminals. Peterson graph (a), results with corrupted (b) and not corrupted graphs structure (c).

7.8. Summary of Experiments and Conclusion

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. 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 an approach which can find and represent important patterns in the chemical domain.

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. Using the evaluation measure without square causes our approach to infer the grammar without one overlapping edge. The evaluation measure with square overcomes this deficiency.

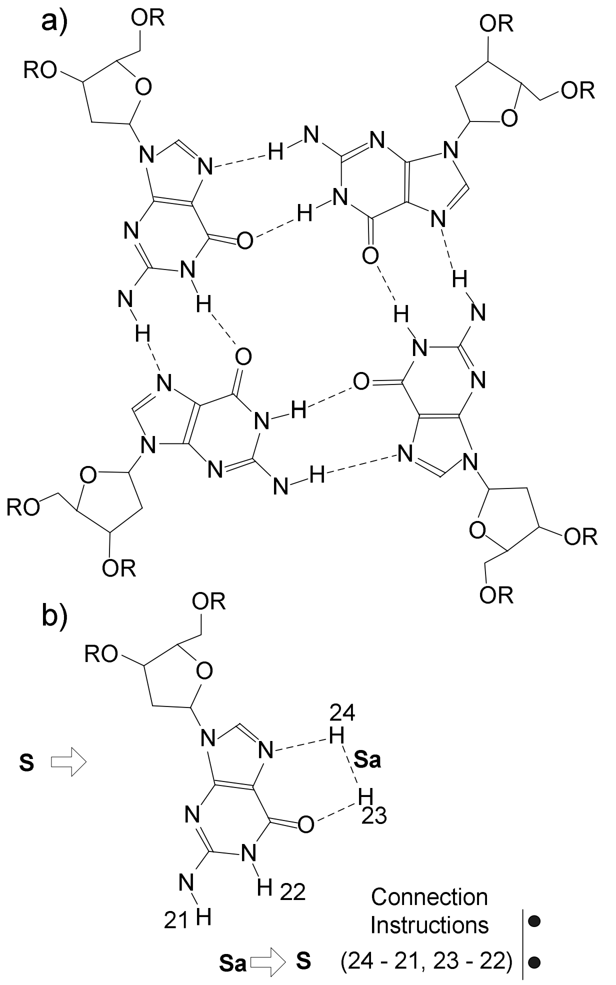


Figure 39: The chemical structure of G tetrad (a) and inferred grammar structure (b).

The approach we described in this dissertation has its limitations. It requires two or more unique labels in the graph, otherwise it infers a two-edge grammar. 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.

CHAPTER 8

INFERRING XML SCHEMA USING GRAPH GRAMMARS

In this chapter we show how our method performs in extracting the organization of XML files. We convert an XML file into a tree and infer a graph grammar from it. We compare the inferred graph grammar to the Document Type Definition of an XML file. We report the graph grammar we found from the XML files used in the National Library of Medicine and the United States Patent and Trademark Office. Our third domain describes a major baseball league.

8.1 Introduction

The World Wide Web Consortium (W3C) released in 1998 the XML recommendation which defines parts of XML document [Harold99]. XML is composed of elements and their attributes. Elements can contain other elements such that they form a hierarchical tree. Software developers often write an XML document and then they write the XML schema or DTD. Several software packages exist which do automatic XML schema generation from XML documenst: Microsoft XSD Inference, Altova XML Spy, EditML Pro, Sonic Software Releases Sonic Stylus Studio and more. These systems are specifically designed to generate XML schema. Our algorithm is more general and is designed to infer graph grammars from any structural data, therefore we do not intend to compete with professional schema generation systems.

Rather, we would like to show an algorithm applicable to many domains and verify it on structures of XML files indicating future direction in analysis of structural data.

An example in Figure 40 shows a graph composed of three overlapping substructures and the graph grammar representation of it. In Figure 41 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.

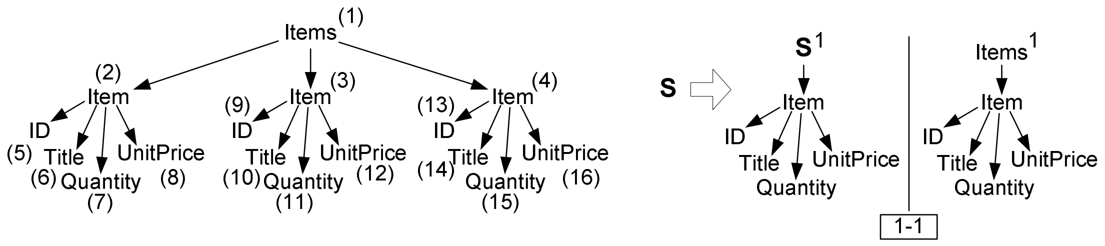


Figure 40: A graph with overlapping substructures and a graph grammar representation of it.

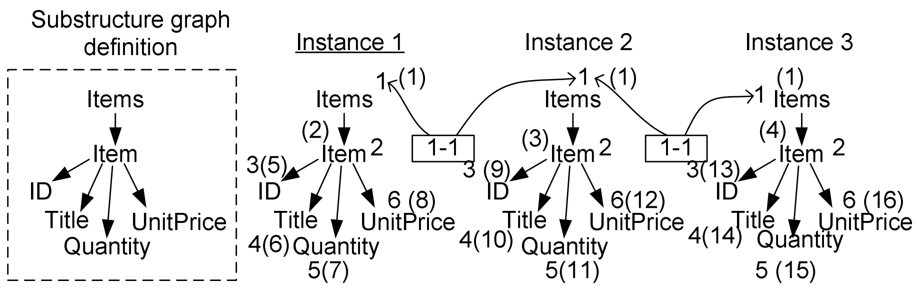


Figure 41: Substructure and its instances while determining connection instructions (continuation of the example from Figure 40).

Every connection instruction has two integers which are indices to the node of a graph used in production. These two integers can refer to arbitrary nodes of the production graph. However, in our experiments with the structure of XML files we encounter often connection 1-1 or 2-2 where the node refers to itself. In this case, in the figures presenting our experimental results we do not draw terminating productions, instead we label one of the nodes S# | X, where X is the terminal node label and S# is s non-terminal label with production number #. For clarity we illustrate it Figure 42.

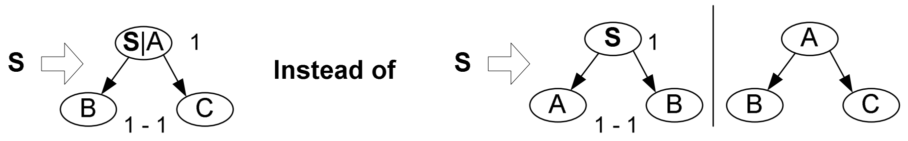


Figure 42: Shorter notation of graph tree grammars where we omit terminating production.

8.2 XML File Conversion to a Graph

We developed a converter which converts an XML file into a tree. We used the Java implementation of the Document Object Model (DOM) in our converter. According to [Ahmed01] 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 the 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 algorithm we need a pattern which is repeated in the graph so we eliminated unique text data (names, card numbers, and price values). We do not assign labels to the edges of the tree.

In Algorithm 5 we show a recursive procedure PRINT_NODE. We call this procedure to convert a DOM tree to a file representing a graph we further use in graph grammar inference. We first call PRINT_NODE procedure with root node of the DOM tree and vertex=1 as input parameters. Then, in line 2, we print to a file the root node of a tree with label Doc. If the input node to the procedure has children and the child is type Element, we execute lines 6 through 9. In these lines we advance an integer which becomes the number of a child node in the newly created graph. We print the node number and its label and then we print the directed edge to this node from parent node. In Figure 43 we show an example of a file where we store the graph after conversion from the DOM tree. The example in Figure 44 corresponds to a graph in Figure 45.

```
vertex 1 DOC
vertex 2 PharmacologicalActionSubstanceSet
directed edge from to 1 2 -
vertex 3 Substance
directed edge from to 2 3 -
vertex 4 RecordUI
directed edge from to 3 4 –
….
```

Figure 43: A file which describes the graph further use to infer graph grammar.

Algorithm 5 Converting DOM tree to a graph file.

```
integer PRINT_NODE (node n, integer vertex)
if (n.getNodeType()==DOCUMENT_NODE)
        print("vertex Number ", vertex, Doc)
parent=vertex;
for (childNode=n.getFirstNode(); childNode≠NULL;
    childNode=childNode.getNextSibling())
        if (childNode.getNodeType()==ELEMENT_NODE)
            vertex++;
            print("vertex ", vertex, childNode.getNodeName())
            print("directed edge from to ", parent, vertex )
            vertex= PRINT_NODE (childNode, vertex)
        end if
end for
return vertex
```

8.3 Domain 1: National Library of Medicine

We selected domains for our experiments where we can easily identify the meaning of the 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) we show in Figure 44. This is a sample file we found on National Library of Medicine website. The tree after conversion of this file we show in Figure 45. In Figure 46 we see graph grammar found by our inference algorithm. Productions S1 and S2 give the representation of a structure of an XML file which contains pharmacological data. Examining 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 DTD with a '+'. We discover these two concepts in production S1 and S2 in Figure 46. However, our inference algorithm 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 the parent to the child nodes of the structure of graphs of S1 and S2 we can find the corresponding DTD entries. For example PharmacologicalActionOfSubstance has only one child DescriptorReferredTo. It corresponds to the DTD entry <!ELEMENT PharmacologicalActionOfSubstance (DescriptorReferredTo)>.

```
<?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
PharmacologicalActionSubstanceSet
(Substance*)>
<!ELEMENT Substance
((RecordUI,RecordName),
PharmacologicalActionList)+>
<!ELEMENT PharmacologicalActionList
(PharmacologicalActionOfSubstance)+>
<!ELEMENT
PharmacologicalActionOfSubstance
(DescriptorReferredTo)>

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

Figure 44: An XML file describing pharmacology data and Document Type Definition (DTD) of it

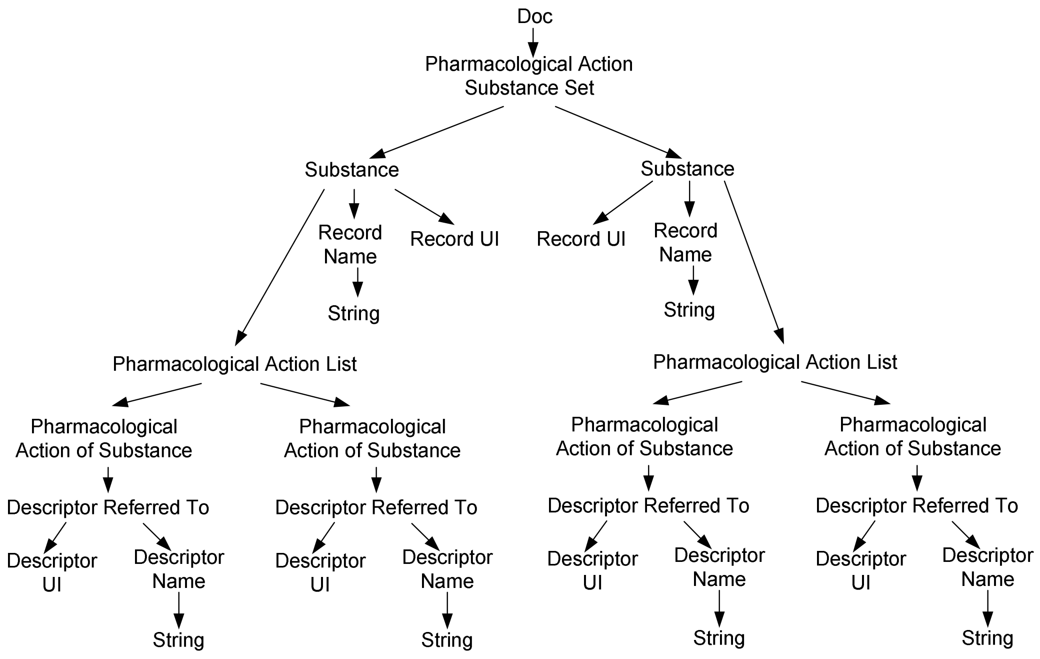
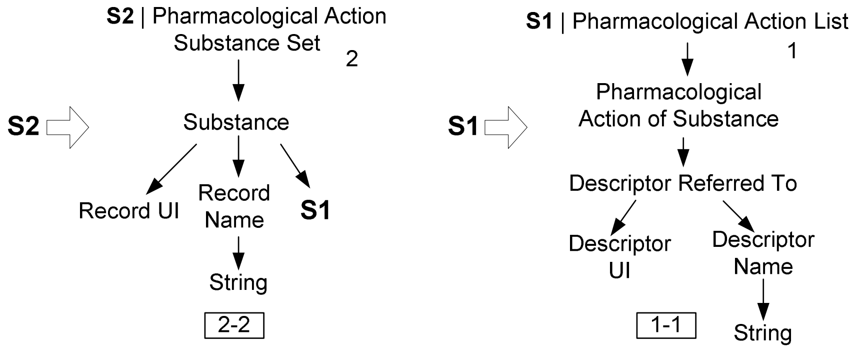


Figure 45: A graph representation of an XML file.



S2 | Pharmacological Action Substance Set
2
S2
Substance
Record UI
Record Name
S1
String
2-2
S1 | Pharmacological Action List
1
S1
Pharmacological Action of Substance
Descriptor Referred To
Descriptor UI
Descriptor Name
String
1-1

Figure 46: Graph grammar found by inference algorithm from the XML tree.

8.4 Domain 2: United States Patent and Trademark Office

In Figure 47 we show a graph grammar inferred from an XML file we found on the United States Patent and Trademark Office website. This is a sample XML file with description of a patent of handheld type four-cycle engine. We converted this XML file to a tree which had 2960 nodes and 2959 edges. Figure 47 shows the first eleven productions inferred from the input tree. They are presented in order. Production S1 has the highest compression value and production S11 the lowest. If we liked to show the complete graph grammar which represents the entire graph with 2960 nodes, we would need to show the remaining graph after its portions are compressed with productions S1-S11. The compressed graph is too big to show.

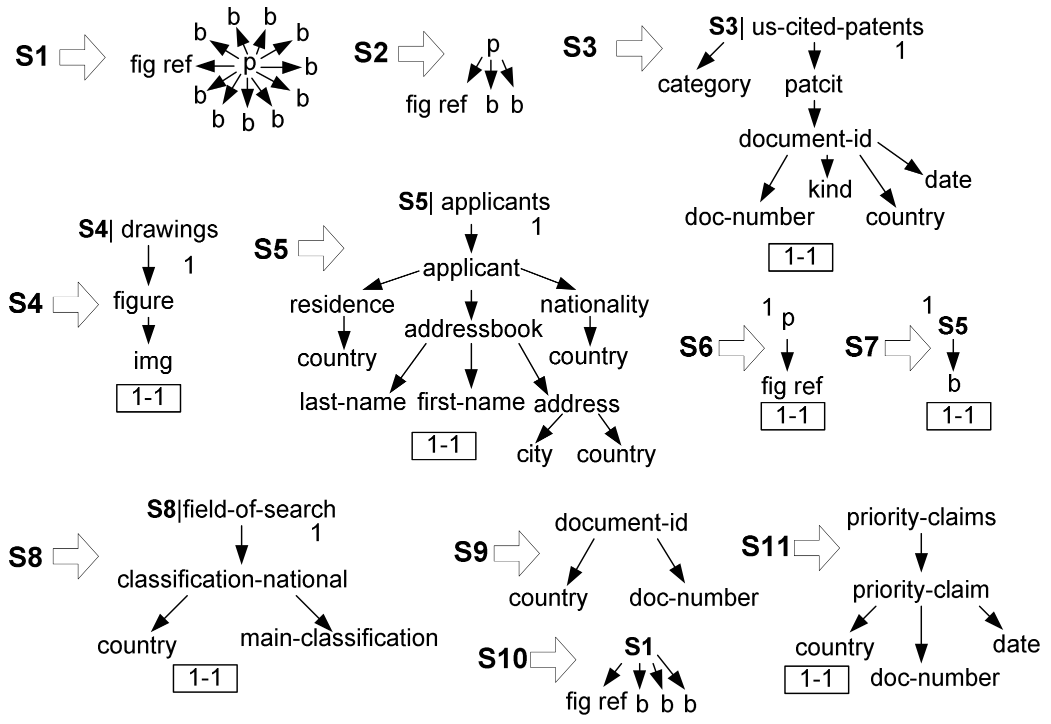


Figure 47: Graph grammar inferred from XML file of a sample patent.

In Table 3 we show selected entries of the DTD for the XML file from which we inferred a graph grammar. We selected these entries to compare with the inferred graph grammar rules in Figure 47. Productions S1, S2, S6, and S10 describe a paragraph. The paragraph is depicted by ‘p’. ‘b’ stands for bold formatting and figref for a reference to a figure. The production S6 is recursive and corresponds to the DTD entry which says that the paragraph can have zero or more figure references. We consider that our inference algorithm found this concept correctly, even though we interpret rule S6 as ‘one or more’ (not ‘zero or more’) of ‘figref’ in ‘p’. DTD also allows for a paragraph to have zero or more bold formatting. Productions S1, S2 and S10 indicate different number of bold formatting in a paragraph accompanied with figure reference but these productions are non-recursive and do not convey the concept of one or more. Therefore, the inferred grammar does not adequately represent the number of bold formatting in a paragraph.

The root of a tree on the right hand side of S3 has two children: category and patcit. S3 is a recursive production. It corresponds to the DTD description where us-cited-patents can have one or more children patcit and category. The child category in DTD is optional. It is indicated by a question mark in Table 3. We do not find any indication in the inferred grammar that the element category is optional. We also did not find us-classification, which is the third optional element of us-cited-patents in DTD. In our current implementation of the graph grammar inference algorithm we do not show explicitly which child node is optional. A post-processing which would show optional child nodes is possible. For instance, the node document-id appears in two productions S3 and S9. S9 has two children country and doc-number. S3 has four children country, doc-number, kind, and date. Comparing the children in these two cases, we can conclude that nodes kind and date are optional children of document-id. It is in agreement with the DTD definition. The element drawings in Table 3 has two alternative elements doc-page+ and figure+. We correctly found that drawings can have one or more figures in production S4. Although our inferred grammar can indirectly indicate alternative elements we did not find doc-page in the inferred graph grammar because this element does not appear in processed XML file.

Table 3: Selected entries of the DTD for patent data

S1, S2, S6, S10	Paragraph	<!ELEMENT p (#PCDATA | b | i | u | sup | sub | smallcaps | br | pre | dl | ul | ol | figref | patcit | nplcit | bio-deposit | crossref | img | chemistry | maths | tables | table-external)*>
S3	Patent citation	<!ELEMENT us-cited-patents (patcit , category? , us-classification?)+>
S4	drawings	<!ELEMENT drawings (doc-page+ | figure+)>
S3, S9	Document identification	<!ELEMENT document-id (country , doc-number , kind? , name? , date?)>
S5	Applicants	<!ELEMENT applicants (applicant+)>
S5	applicant	<!ELEMENT applicant (addressbook+ , nationality , residence , us-rights* , designated-states? , designated-states-as-inventor?)>
S8	Field of search.	<!ELEMENT field-of-search (classification-ipc | classification-national)+>
S11	Priority claim.	<!ELEMENT priority-claim (country , doc-number? , date , office-of-filing? , (priority-doc-requested | priority-doc-attached)?)>
S11	Applications in which priority is claimed.	<!ELEMENT priority-claims (priority-claim+)>

The concept in production S5 that ‘applicants’ contain one or more ‘applicant’ and in production S11 that ‘priority-claims’ contain one or more ‘priority-claim’ is exactly the same as in the DTD in Table 3. The differences between concepts found in the inferred grammar and defined in the DTD result from the limitation of our inference algorithm and from the fact that we used only one XML patent file and we discovered concepts included in this file while the DTD specifies rules for larger set of patents in the United States Patent and Trademark Office which may not be applied in the file in the experiment.

8.5 Domain 3: Major League Baseball

We use an example XML file from [Harold99] which describes the 1998 Major League Baseball season in our experiment in Figure 48. The inferred grammar has six productions. Production S6 is a non-recursive production. Productions S1-S5 are recursive. Production S1 describes the player and it expresses the idea that a team can have one or more players. Production S2 shows that a division can have one or more teams. S3 indicates that a league can have one or more divisions. A season of production S5 can have one or more leagues. We do not show DTD definitions for this domain. Instead, we attempt to write DTD rules based on inferred graph grammar. Based on production S1 we can write

<!ELEMENT Team (Player +)>

<!ELEMENT Player (Surname, Given Name, Position, At Bats, Doubles, Walks, Steals, Errors, Games, Sacrifice Flies, Runs, Sacrifice Hits, Triples, Struck out, Home Runs, RBI, Caught Stealing, Hit by Pitch, Hits, Games Started)>

Based on production S2 we write:

<!ELEMENT Division (Team +)>

<!ELEMENT Team (Team City, Team Name)>

The graph on the right hand side of production S3 contains node S2. In our implementation of the graph grammar inference algorithm we do not have a mechanism which would infer embedding mechanism when production contains a non-terminal node label of another production. Lack of this mechanism prevents us from writing DTD entry based on production S6, S5, S4, and S3. We recognize the need for the above case as a future work. Although we cannot formally write DTD entries when node has non-terminal label of another productions, these production along with the rest of the grammar show what is the structure of data. It shows the frequently accruing recursive motives, the underlying motives of the file.

8.6. Summary of Experiments and Conclusion

We applied algorithm to trees with labels on nodes and directed unlabeled edges. The trees we found by converting an XML file structure. Unique information like names or identification numbers was not part of the processed tree.

We used XML files from three domains in our experiments: pharmacy, patent and baseball. In these domains we found recursive and non-recursive productions. In the pharmacy domain we found two recursive concepts: Pharmacological Action Substance Set contains 'one or more' Substance elements, and Pharmacological Action List contains one or more Pharmacological Action Substance. In the patent domain we found that field-of-search is classified with 'one or more' classification-nationals, drawings contain one or more figures, applicants contains ‘one or more’ applicant elements, priority-claims contain ‘one or more’ priority-claim elements, and us-cited-patents contains ‘one or more’ category and patent citations. In the baseball domain we inferred that a team has one or more players, a division has one or more teams, a league has one or more divisions, and a season has one or more leagues. We inferred these recursive concepts in the form of subgraphs which represent relations for more than two entities we listed above.

We showed that the introduced algorithm of graph grammar inference 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. Indirect detection of alternative or optional elements is possible if in the inferred grammar we find nodes with the same label but different children. The alternative and optional element detection is not part of our implemented algorithm. It remains as a future work.

The method has its limitations. Many of the concepts expressed with the DTD or XML schema we cannot express. For example, we cannot express limits. If a particular item has to accrue two but no more than five times, our inference algorithm can detect it as one or more. We also do not infer an embedding mechanism when one production contains a non-terminal node label of another production. We cannot distinguish between ‘zero or more’ and ‘one or more’ occurrences of the same element. We interpret both of them as ‘one or more’ occurrences. Our graph grammar inference method was created for graphs in general. In this work we applied it to data stored in

XML files. Since we converted XML files to a tree structure, we use our inference algorithm only partially. One can achieve much better results than reported in this work if we would design a schema inference algorithm specifically for XML. Also, the limitation to trees instead of general graphs allows for faster processing and simpler algorithms. However, recently new concepts were proposed related to representing information in the World Wide Web called the Resource Description Framework (RDF). We find that trees are not sufficient to represent some of the RDF concepts and a graph structure with labels on nodes and edges is required [Brickley00]. Our graph grammar inference algorithm can handle such graphs. Performing grammar inference on World Wide Web data described by RDF is our future interest.

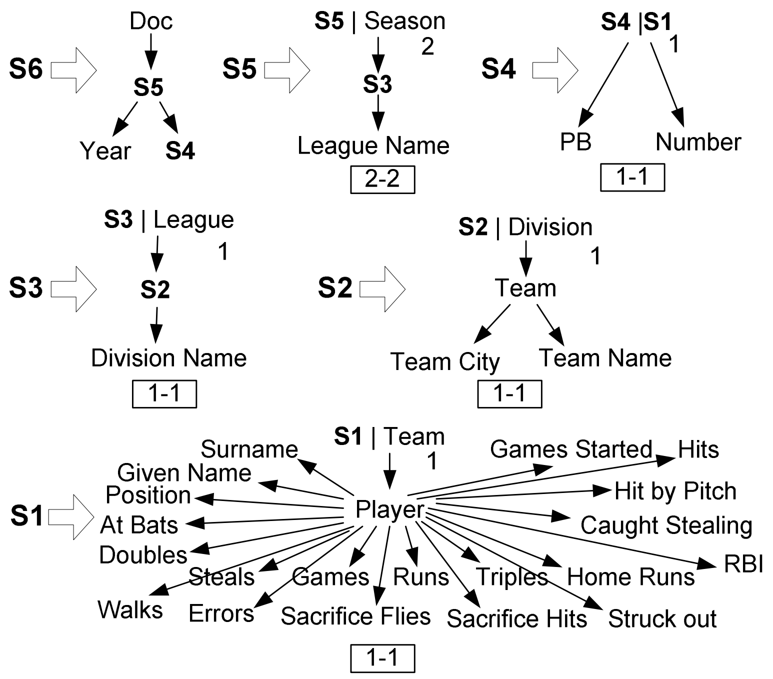


Figure 48: Graph grammar extracted from an XML file of 1998 Major League Baseball season.

CHAPTER 9

INFERRING RECURSIVE PATTERNS IN BIOLOGICAL NETWORKS

This chapter describes experiments on biological networks. After introduction of the database we show learning curves. Then we report experiments with specific biological networks of different species and different networks of the same species. We show how the learning process changes when we increase the size of a sample set. We examine how computation time changes with an increased number of nodes in the input graphs.

9.1 Introduction

The Kyoto Encyclopedia of Genes and Genomes (KEGG) contains graphical representations of cellular processes. We transform this representation to a graph form accepted by our algorithm. The graphs represent processes like metabolism, membrane transport, and biosynthesis. We group the graphs into sets which allow us to search for common recursive patterns which can help to understand basic building blocks and hierarchical organization of processes. There are two reasons for performing experiments in this domain: 1) we wanted to evaluate the algorithm, and 2) we wanted to find graph grammars which represent features of the domain. We perform experiments in two different categories: 1) different biological networks within species, and 2) different species for a particular biological network.

9.2 Experiments with Sets of Different Biological Networks

The biological networks used in our experiments were from KEGG. We use a graph representation which has labels on vertices and edges. The label entry represents a molecule, a molecule group or a pathway. A node labeled entry can be connected to a node labeled type. The type can be a value of the set: enzyme, ortholog, gene, group, compound, or map. A reaction is a process where a material is changed to another material catalyzed by an enzyme. A reaction, for example, can have one or more enzyme entries, and one or more compounds. Labels on edges show relationships between entities. The meanings are: Rct_to_P : reaction to Product , S_to_Rct : substrate to reaction, E_to_Rct : enzyme (gene) to reaction, E_to_Rel : enzyme to relation, Rel_to_E : relation to enzyme. Nodes labeled ECrel indicate an enzyme-enzyme relation meaning that two enzymes catalyze successive reactions.

We use ten species in our experiments. The abbreviated names of the species and their meanings are:

bsu - Bacillus subtilis,

sty - Salmonella enterica serovar Typhi CT18

xcc - Xanthomonas campestris pv. campestris ATCC 33913

pto - Picrophilus torridus

mka - Methanopyrus kandleri

pho - Pyrococcus horikoshii

sfx - Shigella flexneri 2457T (serotype 2a) efa - Enterococcus faecalis

bar - Bacillus anthracis Ames 0581

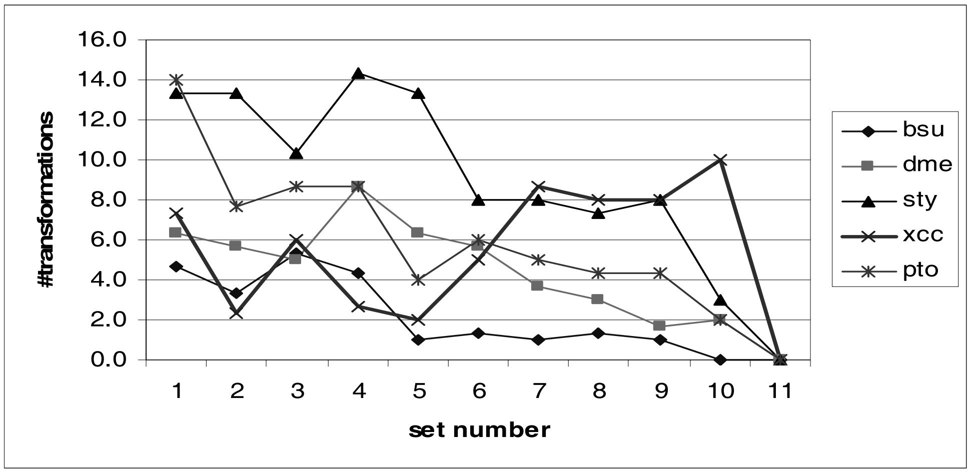
The species we selected randomly from the database. The number of networks is different for each species. We wanted to see how our algorithm performs when we increase sample size of graphs supplied to our inference algorithm. For this purpose we divided all the networks into 11 sets such that the last set (11th) has all the species. Set 10 excludes the 11th portion of all networks. Set 9 excludes 2/11 of all networks and set 1 has 1/11 of all networks. If all networks in the species do not divide by 11 evenly we distribute the remaining networks randomly to the 11 sets.

We would like to compare our inferred grammar from sets of different sizes to the original, true, ideal grammar which represents the species. However, such a graph grammar is not known. In the first experiment we adopted as an original grammar the grammar inferred from the last set. From each set we infer four grammar productions which score the highest in the evaluation. We compute the error (distance) of an inferred grammar to the grammar inferred from the set with all networks. The computation of an error is the same as it is described in section 6.8 on Learning Curves. The error is the minimal number of edges, vertices, and labels required to be change or removed to transform the structure of graph productions from one grammar to the other. In figures we refer to it as #transformations. In Table 4 and in Figure 49 we show the results of the experiment. Every value in the table is an average from three runs. In every run we randomly shuffle the networks over 11 sets such that sets are different in every run. The last column in the table is the average over 11 table entries. Data in Table 5 and Figure 50 is organized in the same way.

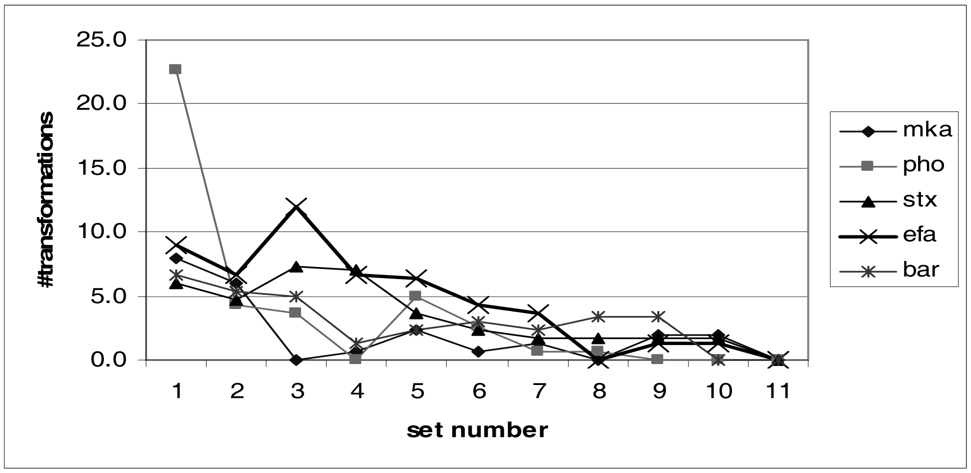
Table 4: Change in inferred grammar from set with increased number of graphs measured as a distance to the grammar inferred from the biggest set.

#Set	bsu	dme	sty	xcc	pto	mka	pho	stx	efa	bar	Average
1	4.7	6.3	13.3	7.3	14.0	8.0	22.7	6.0	9.0	6.7	9.8
2	3.3	5.7	13.3	2.3	7.7	6.0	4.3	4.7	6.7	5.3	5.9
3	5.3	5.0	10.3	6.0	8.7	0.0	3.7	7.3	12.0	5.0	6.3
4	4.3	8.7	14.3	2.7	8.7	0.7	0.0	7.0	6.7	1.3	5.4
5	1.0	6.3	13.3	2.0	4.0	2.3	5.0	3.7	6.3	2.3	4.6
6	1.3	5.7	8.0	5.0	6.0	0.7	2.7	2.3	4.3	3.0	3.9
7	1.0	3.7	8.0	8.7	5.0	1.3	0.7	1.7	3.7	2.3	3.6
8	1.3	3.0	7.3	8.0	4.3	0.0	0.7	1.7	0.0	3.3	3.0
9	1.0	1.7	8.0	8.0	4.3	2.0	0.0	1.7	1.3	3.3	3.1
10	0.0	2.0	3.0	10.0	2.0	2.0	0.0	1.7	1.3	0.0	2.2
11	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

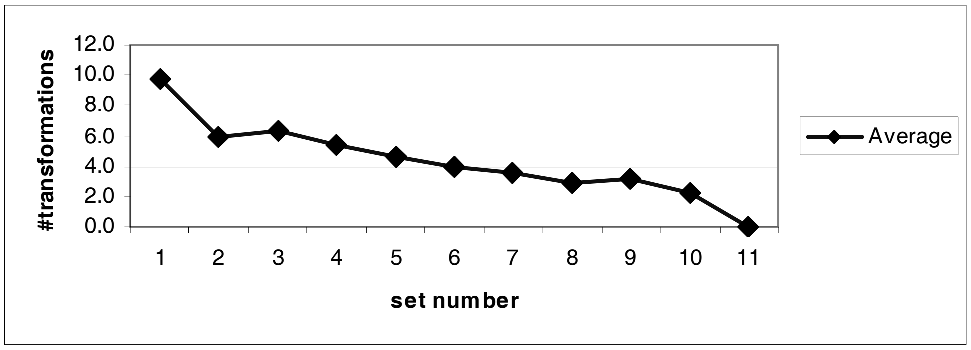
In our next experiment we use a different approach. Instead of comparing the inferred grammar to the grammar inferred from the biggest set, we compare the inferred grammar to the grammar inferred from the next bigger set. The grammar inferred from set 1 we compare to the grammar inferred from set 2, from set 2 to set 3, …, and the grammar from set 10 to the grammar from set 11. We compute the error in the same way as in the last experiment. In Figure 51 we show the graph grammar inferred from a set of thirty and a set of one hundred and ten graphs of Picrophilus torridus (pto).



(a)



(b)

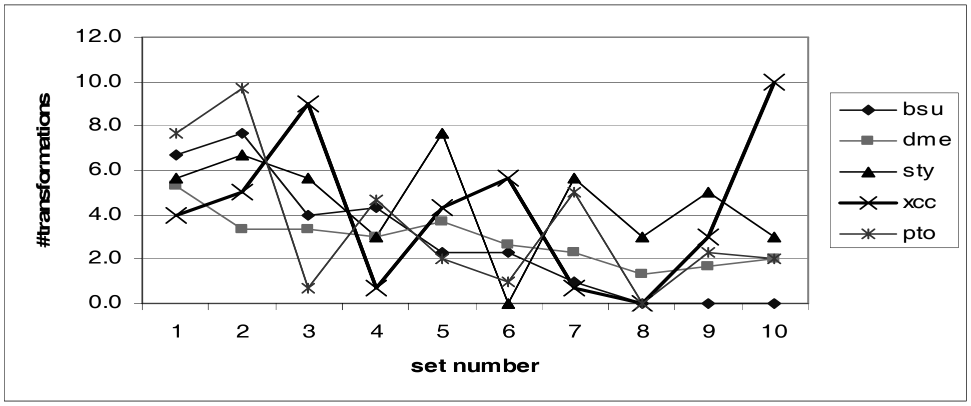


(c)

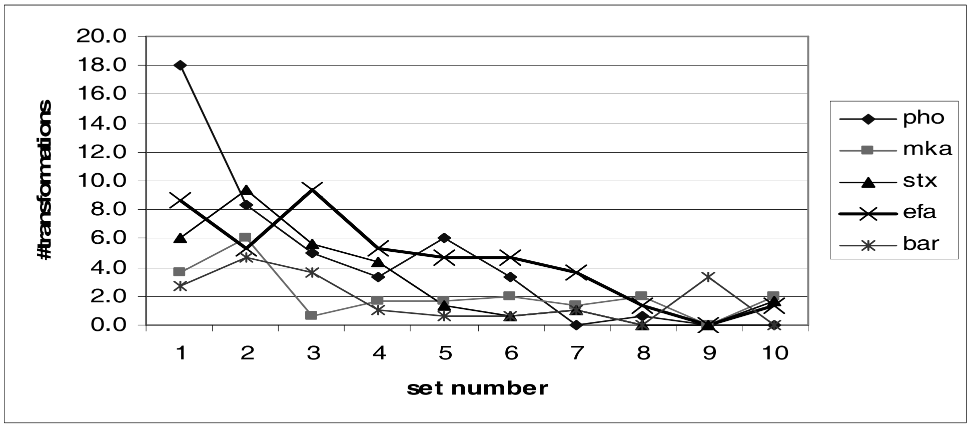
Figure 49: Change in inferred grammar measured in reference to the biggest set in networks of ten species: bsu, dme, sty, xcc, pto (a), mka, pho, stx, efa, bar (b), and average (c).

Table 5: Change in inferred grammar from set of increased number of graphs measured as a distance to the grammar inferred from the next bigger set.

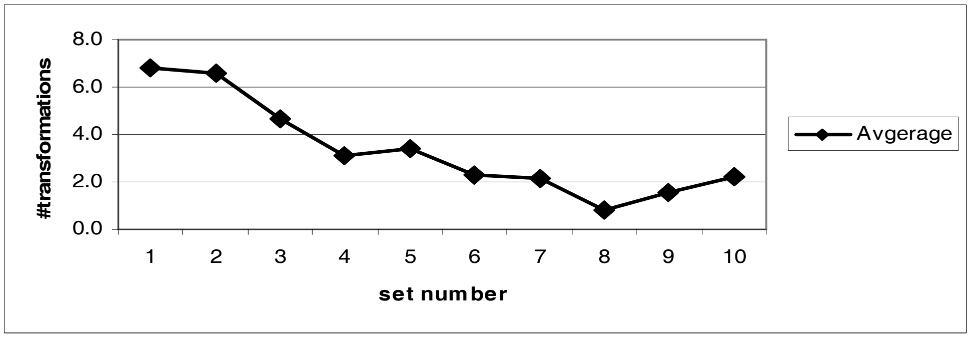
#Set	bsu	dme	sty	xcc	eco	pto	pho	mka	stx	efa	bar	Average
1	6.7	5.3	5.7	4.0	7.7	6.7	18.0	3.7	6.0	8.7	2.7	6.8
2	7.7	3.3	6.7	5.0	9.7	7.7	8.3	6.0	9.3	5.3	4.7	6.6
3	4.0	3.3	5.7	9.0	0.7	4.0	5.0	0.7	5.7	9.3	3.7	4.7
4	4.3	3.0	3.0	0.7	4.7	4.3	3.3	1.7	4.3	5.3	1.0	3.1
5	2.3	3.7	7.7	4.3	2.0	2.3	6.0	1.7	1.3	4.7	0.7	3.4
6	2.3	2.7	0.0	5.7	1.0	2.3	3.3	2.0	0.7	4.7	0.7	2.3
7	1.0	2.3	5.7	0.7	5.0	1.0	0.0	1.3	1.0	3.7	1.0	2.2
8	0.0	1.3	3.0	0.0	0.0	0.0	0.7	2.0	0.0	1.3	0.0	0.8
9	0.0	1.7	5.0	3.0	2.3	0.0	0.0	0.0	0.0	0.0	3.3	1.5
10	0.0	2.0	3.0	10.0	2.0	0.0	0.0	2.0	1.7	1.3	0.0	2.2



(a)

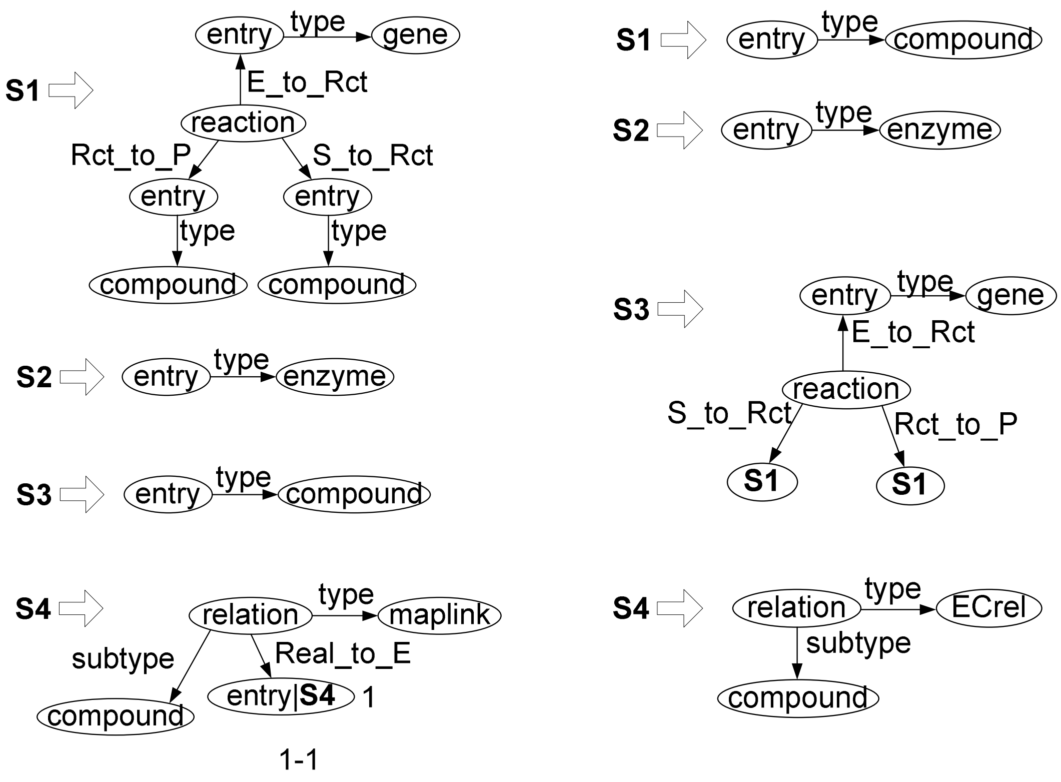


(b)



(c)

Figure 50: Change in inferred grammar measured in reference to the consecutive bigger set in networks of ten species bsu, dme, sty, xcc, pto (a), mka, pho, stx, efa, bar (b), and average (c).



a) b)

Figure 51: Graph grammar inferred from a set of thirty (a) and one hundred and ten (b) graphs of Picrophilus torridus (pto).

9.3 Experiments with Biological Networks from Different Species

In this experiment we construct sets of species with the same biological network. We used ten biological networks in our experiments. The networks' numbers and their meanings are:

10	Glycolysis / Gluconeogenesis
20	Citrate cycle (TCA cycle)
30	Pentose phosphate pathway
51	Fructose and mannose metabolism

61 Fatty acid biosynthesis (path 1)

401 Novobiocin biosynthesis

602 Blood group glycolipid biosynthesis-neolactoseries

730 Thiamine metabolism

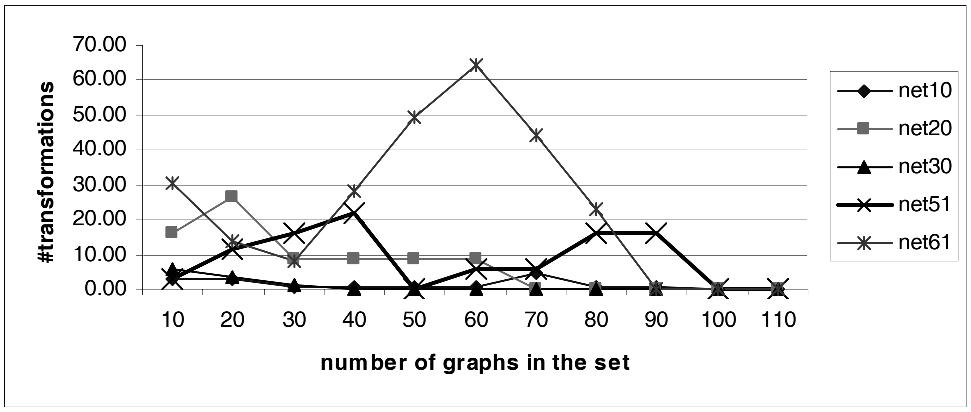
830 Retinol metabolism

930 Caprolactam degradation

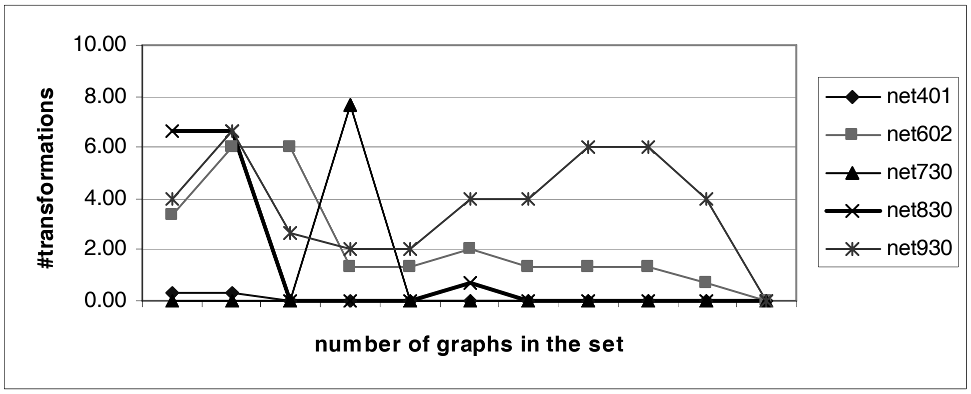
The first experiment in this section is analogous the first experiment of the previous section. In this experiment we examine the change in networks. We created 11 sets. Set number 1 has ten networks, set 2 has twenty networks, and so on. We increase the number of networks in every set by ten such that the last set 11 has one hundred and ten networks. We measure the number of transformations required to transform the grammar inferred from the set to the grammar inferred from set 11 using the method described in the section on Learning Curves. We show results in Table 6 and Figure 52. Every value in the table is an average from three runs. In every run we randomly shuffle the networks over 11 sets such that sets are different in every run. The last column in the table is the average over 11 table entries. In Table 7 and Figure 53 we show results from an experiment where we measure the change from one set to the next bigger set, in the same way as in the previous section. Table 8 and the following Figure 54 show how computation time changes when we increase the size of the input set. We collect how many vertices has the graph created from all graphs in the input set and the time needed for graph grammar inference from the set. In Figure 55 and Figure 56 we show result for network 20 and 30 analogous to results in Figure 54 but with three runs. In every run we randomly shuffle the networks over 11 sets such that sets are different in every run. Figure 57 shows sample graph grammars inferred from the set with ten and seventy graphs of network 10.

Table 6: Change in inferred grammar from set of increased number of graphs measured as a distance to the grammar inferred from the biggest set.

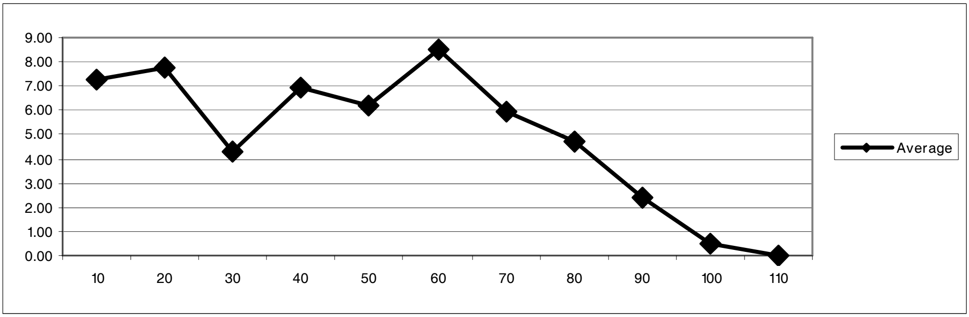
#Set	n10	n20	n30	n51	n61	n401	n602	n730	n830	n930	Avg
1	2.67	16.33	6.00	3.00	30.67	0.33	3.33	0.00	6.67	4.00	7.30
2	2.67	26.33	3.33	11.33	14.00	0.33	6.00	0.00	6.67	6.67	7.73
3	0.33	8.33	1.33	16.33	8.00	0.00	6.00	0.00	0.00	2.67	4.30
4	0.33	8.33	0.00	22.00	28.00	0.00	1.33	7.67	0.00	2.00	6.97
5	0.67	8.33	0.00	0.00	49.33	0.00	1.33	0.00	0.00	2.00	6.17
6	0.67	8.33	0.00	5.67	64.00	0.00	2.00	0.00	0.67	4.00	8.53
7	4.33	0.00	0.00	5.67	44.00	0.00	1.33	0.00	0.00	4.00	5.93
8	0.67	0.00	0.00	16.33	22.67	0.00	1.33	0.00	0.00	6.00	4.70
9	0.33	0.00	0.00	16.33	0.00	0.00	1.33	0.00	0.00	6.00	2.40
10	0.00	0.00	0.00	0.00	0.00	0.00	0.67	0.00	0.00	4.00	0.47
11	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00



(a)



(b)

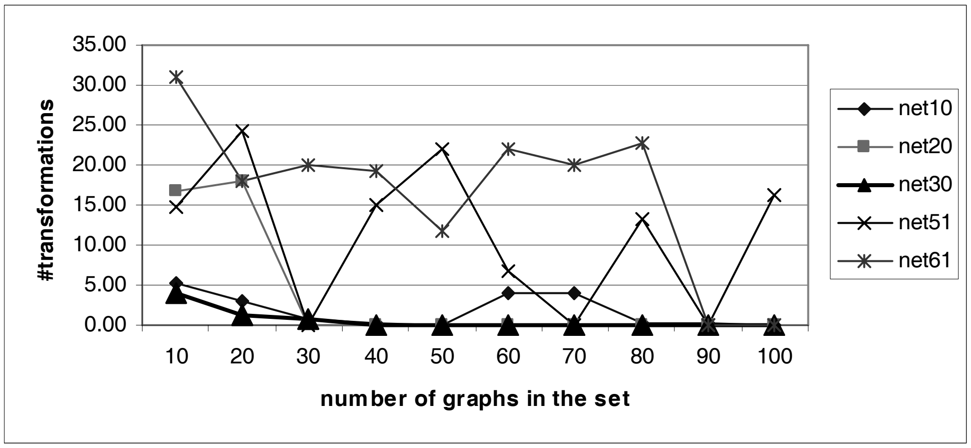


(c)

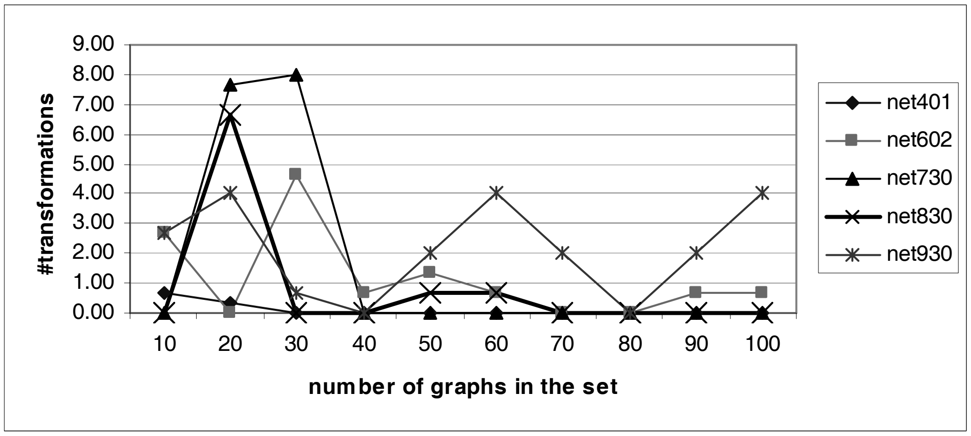
Figure 52: Change in inferred grammar measured in reference to the biggest set in ten networks: network 10, 20, 30, 51, 61 (a), network 401, 602, 730, 830, 930 (b), and average (c).

Table 7: Change in inferred grammar from set of increased number of graphs measured as a distance to the grammar inferred from next bigger set.

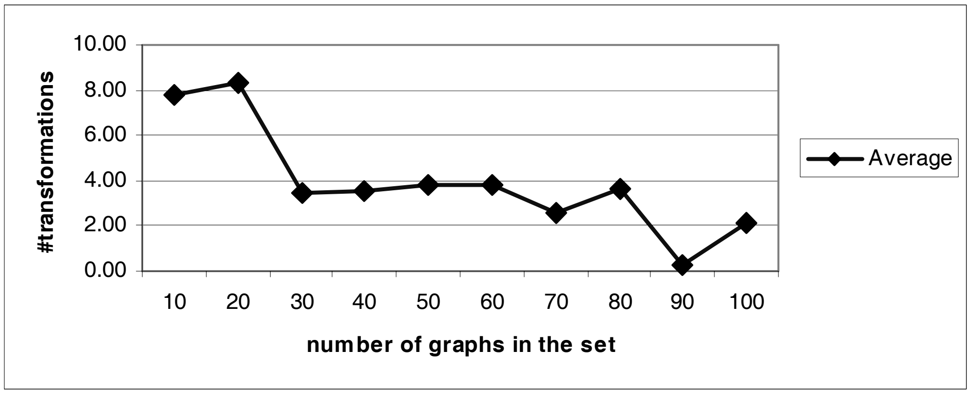
#Set	n10	n20	n30	n51	n61	n401	n602	n730	n830	n930	Avg
1	5.33	16.67	4.00	14.67	31.00	0.67	2.67	0.00	0.00	2.67	7.77
2	3.00	18.00	1.33	24.33	18.00	0.33	0.00	7.67	6.67	4.00	8.33
3	0.67	0.00	0.67	0.00	20.00	0.00	4.67	8.00	0.00	0.67	3.47
4	0.33	0.00	0.00	15.00	19.33	0.00	0.67	0.00	0.00	0.00	3.53
5	0.00	0.00	0.00	22.00	11.67	0.00	1.33	0.00	0.67	2.00	3.77
6	4.00	0.00	0.00	6.67	22.00	0.00	0.67	0.00	0.67	4.00	3.80
7	4.00	0.00	0.00	0.00	20.00	0.00	0.00	0.00	0.00	2.00	2.60
8	0.33	0.00	0.00	13.33	22.67	0.00	0.00	0.00	0.00	0.00	3.63
9	0.33	0.00	0.00	0.00	0.00	0.00	0.67	0.00	0.00	2.00	0.30
10	0.00	0.00	0.00	16.33	0.00	0.00	0.67	0.00	0.00	4.00	2.10



(a)



(b)



(c)

Figure 53: Change in inferred grammar measured in reference to the consecutive bigger set in ten networks: network 10, 20, 30, 51, 61 (a), network 401, 602, 730, 830, 930 (b), and average (c).

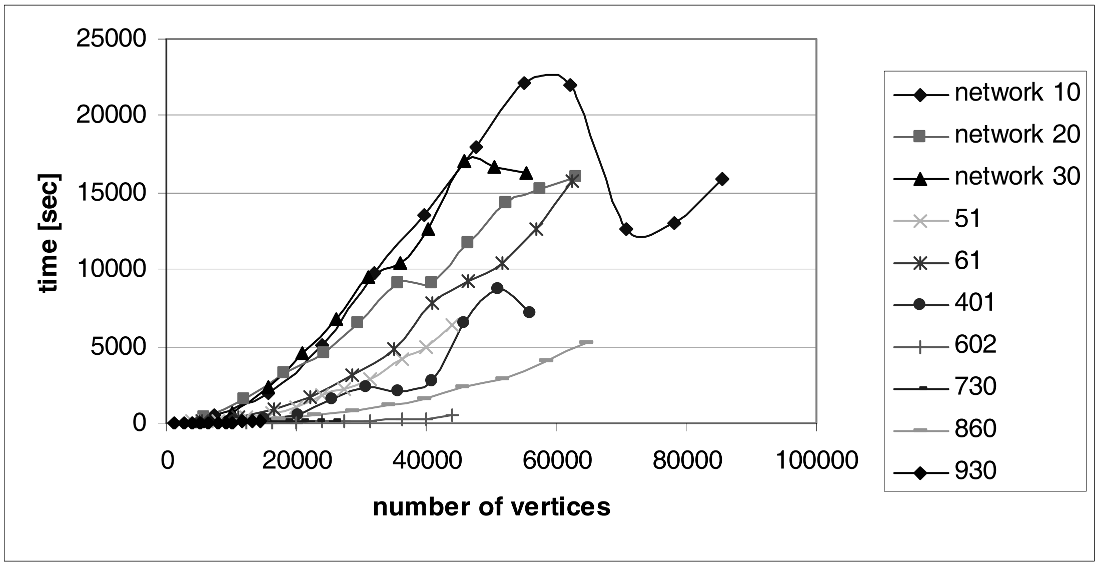
Table 8: Time [sec] of grammar inference as a function of number of vertices in the graph in ten networks.

#V	net10 [sec]	#V	net20 [sec]	#V	net30 [sec]	#V	net51 [sec]
7263	494	5946	395	4882	174	3924	75
15609	1891	12059	1504	10252	609	7848	182
24058	5071	18123	3275	15832	2350	12342	390
32033	9824	24186	4595	20871	4544	16305	700
39799	13565	29662	6569	26074	6724	20139	1027
47625	17985	35694	9055	31103	9547	23955	1881
54978	22087	40821	9120	35894	10357	27528	2182
62252	21973	46372	11769	40414	12566	31314	2839
70621	12601	52167	14308	45980	17104	36348	4123
78004	13081	57627	15264	50562	16672	40068	4893
85428	15919	63111	16035	55431	16332	43953	6413

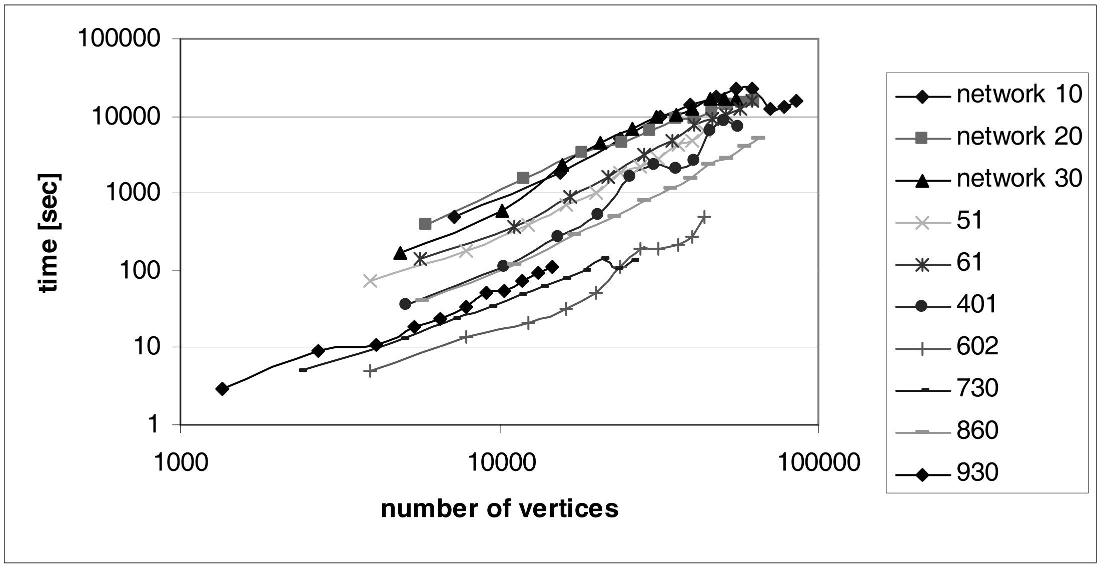
#V	net61 [sec]	#V	net401 [sec]	#V	net602 [sec]
5685	139	5140	35	3924	5
11117	367	10314	114	7848	14
16646	916	15225	274	12342	21
22054	1643	20458	524	16305	32
28603	3177	25577	1624	20139	50
35049	4773	30684	2358	23955	113
40856	7793	35728	2096	27528	190
46366	9254	40787	2702	31314	185
51671	10369	45885	6546	36348	209
56972	12660	50977	8762	40068	272
62463	15772	56033	7138	43953	498

Table 8 - *Continued*

#V	net730 [sec]	#V	net860 [sec]	#V	net930 [sec]
2391	5	5715	40	1346	3
4959	13	11198	117	2722	9
7203	23	17246	286	4098	11
9366	33	22902	491	5381	19
11556	49	28588	779	6565	24
13692	61	34272	1122	7902	33
16098	78	39837	1536	9134	51
18492	100	45394	2310	10306	56
21060	138	51702	2818	11799	72
23385	108	58313	3980	13190	91
25932	130	64494	5195	14581	111

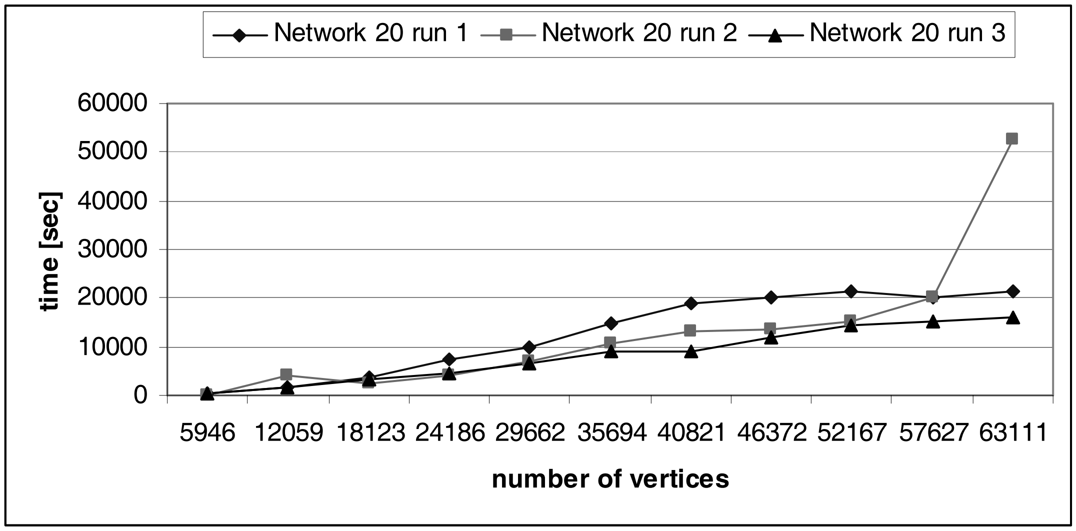


(a)

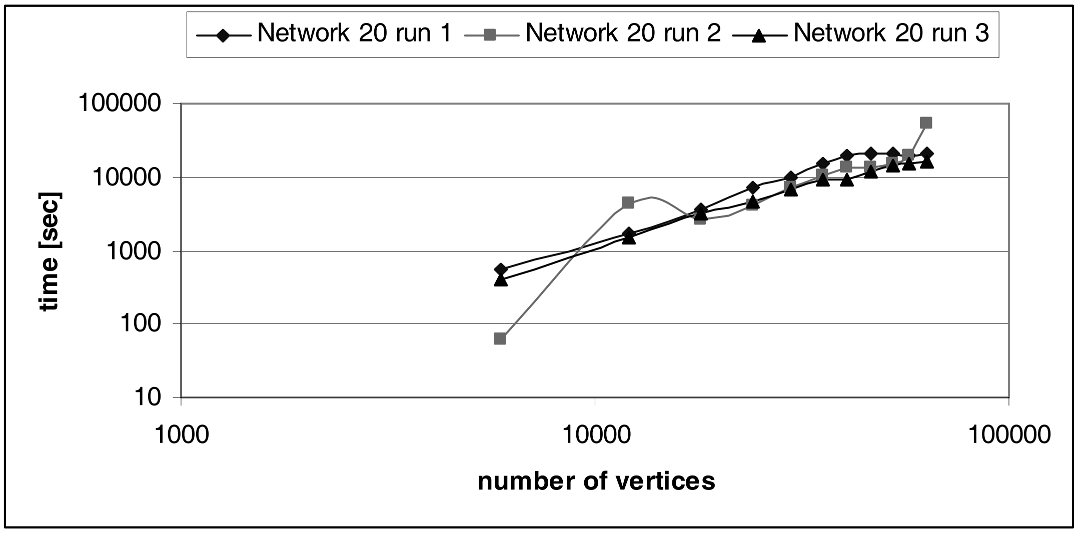


(b)

Figure 54: Time of grammar inference as a function of number of vertices in the graph in ten networks, linear scale (a) and logarithmic scale (b).

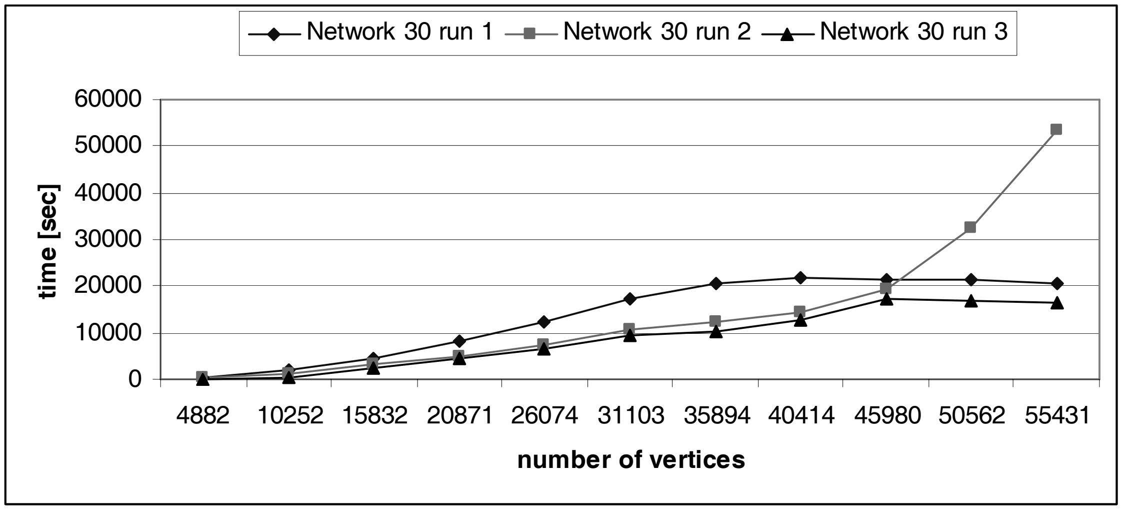


(a)

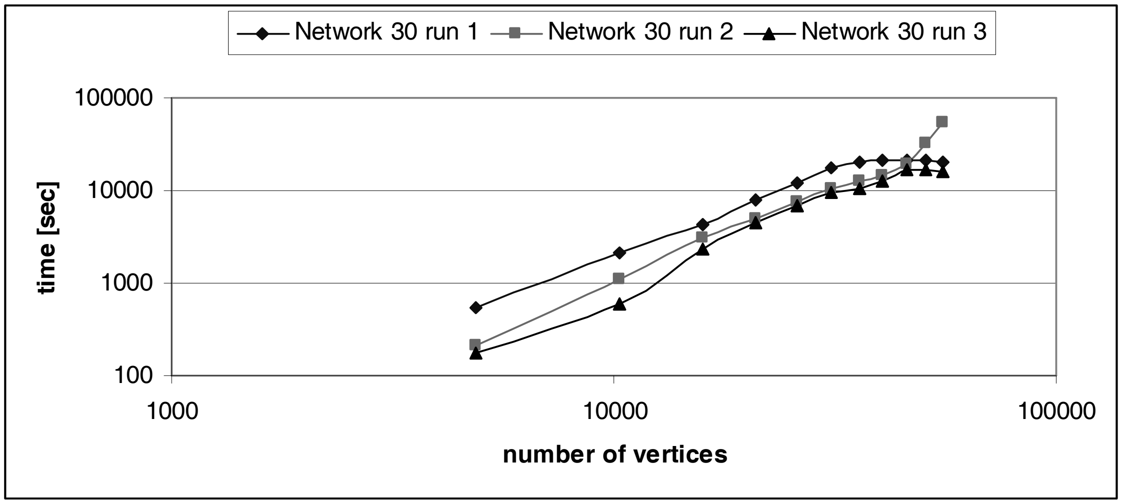


(b)

Figure 55: Time of grammar inference of network 20 in three runs of shuffled species in input sets, linear scale (a) and logarithmic scale (b).

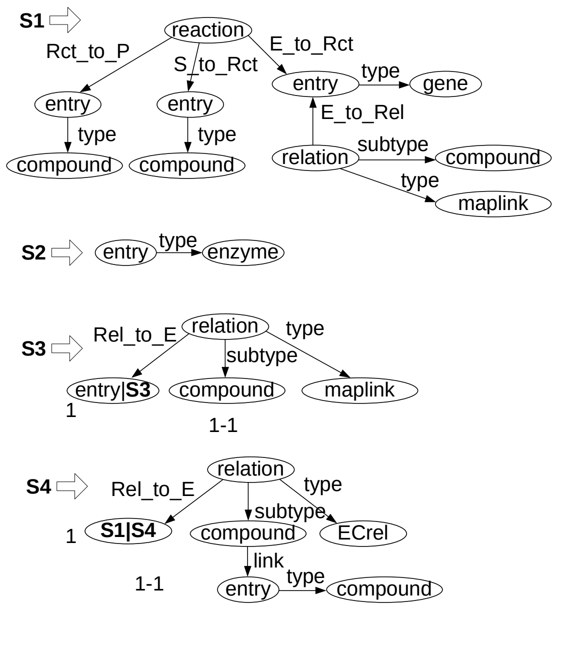


(a)



(b)

Figure 56: Time of grammar inference of network 30 in three runs of shuffled species in input sets, linear scale (a) and logarithmic scale (b).



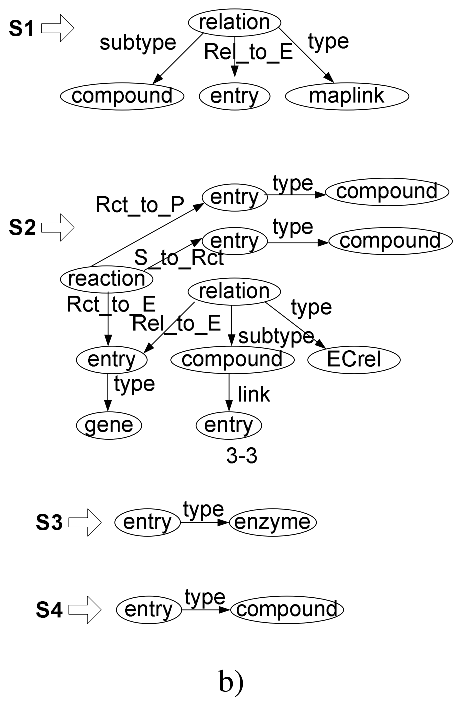


Figure 57: Graph grammar inferred from a set of ten (a) and seventy (b) graphs of network 00010.

The experiments on the biological network domain give us insight into the performance of the algorithm and to the biological networks. Examining Figure 49 we notice that some species, like dme, have a very regular set of biological networks. Increasing the size of the set does not change the inferred grammar. While in other species, like xcc, the set of biological networks is very diverse resulting in significant changes on the curve. Several curves, pto, pho, efa, gradually decrease with the last values being zero. It shows us that our algorithm performed well and with increasing number of graphs in the input set we find the grammar which does not change more with increased number of graphs which indicates that grammar found represents the input set well. The very bottom chart in Figure 49 shows the average change. We see that with the increasing number of graphs in the input sets the curve declines to zero which tells us that with the increasing number of graphs we infer more accurate grammar. We find confirmation of these observations in experiments with sets of biological networks of different species which describe the same process we show in Figure 52. The average change also declines to zero. We see fewer changes in curves in Figure 52 than in Figure 49. It tells us that there is less diversity in set of species within one network than there is in sets of networks within one species.

In Figure 54 we show the computation time as a function of the number of vertices in the input set. We plotted two curves, one in linear, and one in logarithmic scale. The curves in linear scale become almost straight lines in logarithmic scale which confirms experimentally the polynomial complexity of the algorithm. Time curves of network 10, 30, and 401 have a surprising dip towards the right end of the scale where we would expect an increase in computation time, but instead observe a decrease. We suspected that it is because in these cases the input set of graphs gets compressed very well in iteration one or two of grammar inference and the compressed graph used in iterations three and four is small which results in faster execution time. However, a closer look at the number of vertices in each iteration did not confirm this. Since the isomorphism test and the heuristic used in the algorithm have the main influence on the computation time, we tend towards relating the decreasing time phenomenon to these features of the algorithm.

CHAPTER 10

CONCLUSIONS AND FUTURE WORK

In this dissertation we have studied algorithms for inferring node and edge replacement graph grammar. The algorithm starts from all nodes with the same label and grows them by adding to them one node or node and an edge at a time. We developed a substructure which consists of the definition of a graph and all subgraphs appearing in the input graph that are isomorphic to this graph definition (i.e., instances). Every time we add a new node to the substructure we check if instances overlap on one node. The overlap of instances proposes a recursive graph grammar production which expresses concepts of ‘one or more’ of the same substructures. The input graph to our algorithm is an arbitrary directed or undirected graph with labels on nodes and edges.

We described two algorithms for inference of two classes of graph grammars. The first class of graph grammars we call Node Replacement Recursive Graph Grammars. The second we call Edge Replacement Recursive Graph Grammars. The algorithms are based on previous work in frequent substructure discovery. We used frequently occurring, highly compressing, subgraphs as a guide to find the basic building blocks of the input graphs. The algorithms check if frequent subgraphs overlap by a node (node replacement) or two nodes (edge replacement) and propose a recursive graph grammar production if they do.

The node replacement recursive graph grammar inference algorithm limits productions to one single node on the left hand side. The algorithm infers either recursive or non-recursive productions depending if frequent subgraphs in the input graphs overlap or not. Smaller inference error occurs when the inferred pattern has higher MDL value, i.e., is more complex. We infer a graph grammar with only one edge because it is evaluated the highest when nodes and edges in the graph are labeled with only one label. Our approach can infer Jonyer et al.'s class of grammars.

Our approach can find embedding mechanism for recursive productions in the form of connection instructions. 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. Adding a precise embedding mechanism for non-recursive productions would require reference to the nodes of the compressed portion of the graph and the remaining uncompressed portion of the graph. It means that we would not be minimizing the description length of a graph with the compressed substructure but enlarging it. Another approach to specifying connection instructions for non-recursive productions would be to allow for a less precise mechanism where we can mark nodes that connect the substructure to the reset of the graph and give labels of vertices in the uncompressed portion to which the nodes are connected. This method might be useful in some applications where this information is important but would not allow for regeneration of the structure of the input graph from the inferred grammar.

In our approach we infer one production in each iteration. The one time compression pass on the input graph prevents us from learning alternating productions, because the inferred production has no way to refer to productions in future iterations. We could infer alternating productions using two different approaches: 1) do multiple compression passes on the input graph or 2) search for multiple productions in one iteration.

We proposed inference of edge replacement recursive graph grammars as an extension to the algorithm for node replacement inference. We allowed for overlap by two nodes and we inferred grammars with a *real* or *virtual* edge. 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, therefore, we have an approach which can find and represent important patterns in the chemical domain. 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. Adding edges to the inferred pattern increases the error. The highest error we had is with a complete graph.

We used XML files from three domains in our experiments: pharmacy, patent and baseball. In these domains we found recursive and non-recursive productions. We showed that the introduced algorithm of the graph grammar inference 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. Indirect detection of alternative or optional elements is possible if in the inferred grammar we find nodes with the same label but different children. Alternative and optional element detection is not part of our implemented algorithm. It remains as a future work. Performing grammar inference on World Wide Web data described by RDF is also a future direction in this domain.

In experiments with biological networks we notice that some species, like dme, have a very regular set of biological networks. Increasing the size of the set does not change the inferred grammar. While in other species, like xcc, the set of biological networks is very diverse. Several curves (pto, pho, efa), which represent the change in error with the increased sample set, gradually decrease, with the last values being zero. It shows us that our algorithm performed well and with an increasing number of graphs in the input set we find the grammar, which does not change more with an increased number of graphs, which indicates that the grammar found represents the input set well. The computation time curves we plotted in linear scale become almost straight lines in logarithmic scale, which confirms experimentally the polynomial complexity of the algorithm.

We would like to indicate general future directions in graph grammar inference research. They are:

(1) Integration of inference of non-recursive, node-replacement and edge-replacement productions into one graph grammar inference algorithm. The user of the algorithm might be interested in the best recursive or non-recursive graph pattern which describes the data. The most interesting pattern can be the one which scores the highest in the evaluation process. We can achieve the integration by placing candidate patterns

on the queue and let non-recursive, recursive node and edge replacement productions compete against each other.

(2) Develop algorithms which allow for learning larger classes of graph grammars. In this dissertation we extended classes of presently learnable graph grammars. It is possible to extend it even further into context sensitive graph grammars where we could still replace nodes and edges, but whether or not this replacement takes place depends on the neighborhood of the replaced node or edge. In order to regenerate structures we would need more sophisticated generation mechanism with a context sensitive embedding mechanism. This mechanism, inferred during induction, would indicate nodes to merge during the generation process. We can explore other techniques like decomposition of graphs in searching for the best grammar which describes the data.

(3) Investigate learnable properties of graphs from the perspective of graph grammars. Planarity or average degree of nodes are standard properties of graphs, and algorithms exist for determining whether these properties. With graph grammars we can check other properties, for example the existence of recursive patterns, repetitive patterns, hierarchical patterns or overlapping patterns.

(4) Identify experimental areas and show the significance of graph grammar inference in these domains. One of the new domains we approach is visual languages, where graph grammar inference from the sample of a language can give a grammar to be used to check newly written programs.

(5) Use graph grammar inference to identify building blocks, modularity and motifs in biology, software, social networks, and electronics circuits. We did experiments in biology and XML domains. Biological and chemical structures are still very promising areas of the application of recursive graph grammars. Social networks, Very Large Scale Integrated circuits, and the Internet are domains with relational data whose hierarchy and recursive properties we can explore with graph grammars.

(6) Expand graph grammar inference to learning stochastic graph grammars. This extension would require assigning a probability to each production. We can evaluate this probability based on the portion of the input graph covered by the inferred production.

(7) Developing a better error measure for evaluating inferred graph grammars. In our experiments we measured an error based on structural difference. Another approach to measuring is the accuracy of the inferred grammar would be based on a graph grammar parser. We would consider accurate the inferred grammars that can parse the input graph. Graph grammar parser would require subgraph isomorphism test which is computationally expensive and much more difficult in implementation than the error measure we are using.

REFERENCES

Bunke83	H. Bunke, G. Allermann, *Inexact graph matching for structural pattern recognition.* Pattern Recognition Letters, 1(4) 245-253. 1983
Carrasco01	Rafael C. Carrasco, Jose Oncina and Jorge Calera-Rubio, "Stochastic Inference of Regular Tree Languages," Machine Learning 44, 185--197, 2001.
Chittimoori99	R. Chittimoori, L. Holder, and D. 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.
Chomsky56	Noam Chomsky, *Three models of language*. IRE Transactions in Information Theory 2, 3, 113-24, 1956
Cook00	D. Cook and L. Holder, *Graph-Based Data Mining.* IEEE Intelligent Systems, 15(2), pages 32-41, 2000.
Cook94	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
Doshi02	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.

Drewes90 F. Drewes, H. Kreowski, *A note on hyperedge replacement grammars*. Lecture Notes in Computer Science 532. Graph Grammars and their application to computer science. 1990.

Flasinski98 M. Flasinski, *Power properties of NLC graph grammars with a polynomial membership problem,* Theoretical. Computer. Science., 201(1-2), 189-231, 1998.

Fletcher01 Peter Fletcher. *Connectionist learning of regular graph grammars*. Connection Science, 13, no. 2, 127-188. 2001

Fu1 King-Sun Fu, T. L. Booth. *Grammatical inference: introduction and survey—part I* . IEEE Transactions on Pattern Analysis and Machine Intelligence archive Volume 8 , Issue 3 , 343 - 359 ,1986

Fu2 King-Sun Fu, T. L. Booth. *Grammatical Inference: Introduction and Survey|Part II.* IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. PAMI-8, no. 3: 360-375, May 1986.

Gernert97 D. Gernert, *Graph grammars as an analytical tool in physics and biology*. Biosystems 1997, vol. 43, no. 3, pp. 179-187(9), 1997

Gold E Mark. Gold. *Language identification in the limit.* Information Control, 10: 447-- 474, 1967.

Gonzalez00 J. A. Gonzalez, L. Holder, and D. Cook. *Structural knowledge* *discovery used to analyze earthquake activity*. In Proceedings of the Thirteenth Annual Florida AI Research Symposium, 2000.

Habel92 A. Habel, *Hyperedge Replacement: grammars and Languages*. Lecture Notes in Computer Science. 643. 1992.

Hopcroft79 John Hopcroft, Jeffrey Ullman, *Introduction to Automata Theory*, Languages, and Computation, Addison-Wesley, 1979

ICGI00 Proceedings of the 5th International Conference on Grammatical Inference (ICGI 2000), © Springer-Verlag, LNCS 1891

ICGI02 Grammatical Inference: Algorithms and Applications; 6th International Colloquium, ICGI 2002, volume 2484 of LNCS/LNAI. © Springer-Verlag, 2002

ICGI94 Grammatical Inference and Applications, Second International Colloquium, ICGI94, Alicante, Spain, September 21-23, 1994, number 862 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1994.

ICGI98 Grammatical Inference, ICGI98 , number 1433 in Lecture Notes in Artificial Intelligence,. Springer Verlag, 1998

Janssens80 D. Janssns, G. Rozneberg, *On the structure of node-label-controlled graph languages*, Information. Science. 20 (1980) 191-216

Janssens82 D. Janssens, G. Rozenberg, *Graph grammars with neighborhood-controlled embedding*. Theoretical Computer Science 21, 55-74, (1982)

Janssens83 D. Janssens, G. Rozenberg, *Graph grammars with node-label* *controlled rewriting and embedding*. Lecture Notes in Computer Science, vol. 153, Springer, Berlin, 186-205. 1983

Jeltsch90 E. Jeltsch, H. Kreowski, *Grammatical Inference Based on Hyperedge Replacement. Graph-Grammars.* Lecture Notes in Computer Science 532, 1990: 461-474, 1990

Jonyer02 I. Jonyer, L. Holder, and D. Cook, *Concept Formation Using Graph Grammars*, Proceedings of the KDD Workshop on Multi-Relational Data Mining, 2002.

Jonyer04 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.

Jonyer04 I. Jonyer and L.B. Holder, and D.J. Cook, *MDL-Based Context-Free Graph Grammar Induction and Applications*. International Journal of Artificial Intelligence Tools, Volume 13, No. 1 pages 65-79, 2004.

Kim97 C. Kim, *A hierarchy of eNCE families of graph languages.* Theoretical Computer Science 186, 157-169, 1997.

Kreowski90 Eric Jeltsch, Hans-Jörg Kreowski, *Grammatical Inference Based on Hyperedge Replacement.* Graph-Grammars and Their Application to Computer Science. Lecture Notes in Computer Science 532, 461-474, 1991

Kuramochi01 M. Kuramochi and G. Karypis, *Frequent subgraph discovery.* In

Proceedings of IEEE 2001 International Conference on Data Mining (ICDM '01), 313-320, 2001.

Lari91	K. and S.J. Young. *Applications of stochatic context-free grammars using the inside-outside algorithm*. Computer, Speech and Language, pages 237--257, 1991.
Lopez98	Damián López, Jose M. Sempere. *Handwritten Digit Recognition through Inferring Graph Grammars. A First Approach*. Lecture Notes In Computer Science, Proceedings of the Joint IAPR International Workshops on Advances in Pattern Recognition, Pages: 483 – 491, Springer-Verlag 1998
Mehta03	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.
Milo02	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
Nevill-Manning97	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

Oates03	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.
Read98	R Read and R. Wilson, *An Atlas of Graphs*. Oxford University Press, 1998
Rozenberg86	G. Rozenberg, E. Welzl, Boundry NLC Graph Grammars – Basic Definitions, Normal Forms, and Complexity. Information and Control, 69, 136-167 (1986)
Rozenberg97	G. Rozenberg (Ed.), *Handbook of Graph Grammars and Computing by Graph Transformations*, Volume 1: Foundations. World Scientific 1997
Sakakibara92	Yasubumi Sakakibara. *Efficient learning of context-free grammars from positive structural examples.* Information and Computation, 97:23--60, 1992.
Sakakibara97	Y. Sakakibara, *Recent advances of grammatical inference*. Theoretical Computer Science, 185:15-45, 1997.
Sanchez01	Gemma Sánchez, Josep Lladós, and Karl Tombre. *An Error-Correction Graph Grammar to Recognize Textured Symbols.* In 4th IAPR

	International Workshop on Graphics Recognition, Kingston, Ontario, Canada, pages 135-146, September 2001.
Schumann03	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
Su99	S. Su, D. Cook, and L. Holder. *Knowledge discovery in molecular biology: Identifying structural regularities in proteins.* Intelligent Data Analysis, 3, 413-436, 1999.
Uhl04	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
Yan02	X. Yan and J. Han, *gSpan: Graph-based substructure pattern mining.* In IEEE International Conference on Data Mining, Maebashi City, Japan, 2002.
Zhang03JK	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

Kukluk06	J. Kukluk, L. Holder and D. Cook, *Inference of Node Replacement Recursive Graph Grammars*, Proceedings of the SIAM Conference on Data Mining, 2006.
Brickley00	D. Brickley, Rdf Specifications, *Containing Resource Description Framework Rdf Schema and Resource Description Framework Rdf Model and Syntax Specification*, Iuniverse Inc, 2000
Harold99	E. Harold, *XML Bible*, Hungry Minds (1999)

BIOGRAPHICAL INFORMATION

Jacek Kukluk was born in Poland in 1974. After completion of graduate study in electrical engineering at The University of Zielona Gora (Poland) in 1999 he worked for this university as a teacher and researcher. He taught an introductory computer programming course and lectured in laboratories. His research in Poland focused on power electronics. He participated in projects funded by the Committee of Science Research of Poland related to power quality and electrical energy conversion using power electronic devices. In 2000, Stefan Batory Foundation awarded for Jacek Kukluk a scholarship. Thanks to this support he was able to come to the Energy Systems Research Center at University of Texas at Arlington as a visiting scholar. While working with researchers in power systems, he became interested in obtaining a degree in computer science. In 2003, he completed his Master of Science in Computer Science and Engineering at The University of Texas at Arlington. In 2006, at the same university, he defended his dissertation and received Doctor of Philosophy in Computer Science and Engineering.