Constructing VEGGIE: Machine Learning for Context-Sensitive Graph Grammars

Keven Ates, Kang Zhang
University of Texas at Dallas
atescomp@utd.edu, kzhang@utd.edu

Abstract

Context-sensitive graph grammar construction tools have been used to develop and study interesting languages. However, the high dimensionality of graph grammars result in costly effort for their construction and maintenance. Additionally, they are often error prone. These costs limit the research potential for studying the growing graph based data in many fields. As interest in applications for natural languages and data mining has increased, the machine learning of graph grammars poses a prime area of research. A unified graph grammar construction, parsing, and inference tool is proposed. Existing technologies can provide a context-free tool. However, a general context-sensitive tool has been elusive. Using existing technologies for graph grammars, a tool for the construction and parsing of context-sensitive graph grammars is combined with a tool for inducing context-free graph grammars. The system is extended with novel work to infer context-sensitive graph grammars.

1. Introduction

Well-established construction and parsing technologies currently exist in the field of context-sensitive graph grammars. To a lesser extent, graph grammars induction technologies also exist, most of which focus on context-free applications. Currently, only one known induction system exists for the inference of a limited context-sensitive graph grammar [2]. However, no known tool exists that combines both of these technologies into a unified system. This fact is also complicated by the lack of a widely accepted standard for a common structure for graphs, much less a standard for graph grammars. This paper addresses these issues by presenting VEGGIE, a Visual Environment for Graph Grammar Induction and Engineering.

VEGGIE is a tool that marries an existing parsing system with an existing induction system and is extended with novel work for context-sensitive induction. Therefore, the tool represents a reinforcement learning system composed of and extending existing technologies. While the concept of such a tool is a simple idea, constructing a tool from existing technologies presents its own challenges, particularly in the areas of structure compatibility, robustness, and space-time efficiencies.

There are two primary motivations for creating such a system. The first focuses on reducing the manpower involved in producing complex graph grammars for visual languages. The manpower requirements for constructing a graph grammar can be costly for many languages. This has had a detrimental effect on the acceptance of graph grammars in mainstream computing. By constructing examples of a visual language as a learning set, a preliminary graph grammar can be quickly constructed by a machine learning system. Language examples are an intuitive way for a designer to represent a language since foreseen complexities are generally modeled with existing systems such as the UML. Examples also lower the learning curve associated with a user's understanding of a new language. Unforeseen complexities can be added later through an editing process or modeled and added to the learning set for reconstruction of the grammar.

The second motivation is due to the increasing interest in mining large bodies of data. As databases can be modeled as graphs, a graph parsing and induction system can infer a grammar from the given data exposing hidden structures. The inferred grammar can also be used to parse related databases. The parsing system could report which structures exist in other databases. A parsing failure could result in a reinforcement learning for new structures or expose embedded data aberrations requiring further analysis.

2. A graph grammar system

A visual language can be defined as “a pictorial representation of conceptual entities and operations and is essentially a tool through which users compose

visual sentences" [3]. Graph grammars are central to the study of visual languages. As such, a brief history of an existing graph grammar system gives some background for the requirements of VEGGIE.

Graph grammar systems have existed for at least two decades. Some of the earliest work on visual languages date back to the 1970s [4, 5]. Some foundational work even dates back to the 1960s [6]. However, interest in graph grammar technology began growing in the 1980s with significant enhancements in the 1990s. The history and progress of one graph grammar system provides the basis for VEGGIE. The development of the Reserved Graph Grammar and its derivatives are presented as foundational works. The Spatial Graph Grammar derivative is presented as a state-of-the-art system for graph grammar development and is the basis for VEGGIE.

2.1 The Reserved Graph Grammar

An important advancement in graph grammars was realized with the development of the Reserved Graph Grammar (RGG) [8]. This work is based on prior work from the Layered Graph Grammar (LGG) [7]. LGG provided two important advancements in graph grammar development. Most graph grammars system only address context-free implementations. Context-free graph grammars allow for only a single non-terminal in the left hand side of a production rule. LGG provided a context-sensitive environment allowing for a more robust expressiveness for the grammars. Context-sensitive graph grammars allow for an arbitrary number of nodes and edges in either side of a production rule. Additionally, LGG provided an intuitive graph formalism for designing grammars. Each production rule in a grammar consists of left hand side (LHS) graph and a right hand side (RHS) graph. However, the parsing system was inefficient. The graph membership problem is generally NP-hard. As an initial advancement for graph grammars into context-sensitivity, the LGG parser often reached exponential time. Improvements to LGG led to the development of RGG.

The RGG system extends the LGG formalism with an embedding mechanism that enhances nodes with visual notation resulting in a simplified grammar representation. The simplification allows the resulting grammar to avoid much of the context specification required in LGG. This increases the expressiveness of the grammar as well as the efficiency of the parsing algorithm.

Given this efficiency, a general parsing algorithm for RGG is still exponential in time. RGG improves on this by introducing a constraint on the grammar. The *confluence constraint* [8] restricts the parser to grammars that do not rely on the order of selection for production rules to validate a graph—the *selection-free* condition [8] . The selection-free condition states that any failed parsing path indicates all other paths also fail and backtracking in the parser is not required. This reduces the time complexity of the parser to polynomial time [8]. The confluence constraint's effect on languages is unknown. However, empirical tests indicate that practical applications are robust. The requirement that a graph grammar be selection-free results in a confluence checking algorithm which is based on the well known Critical Pair lemma and Church-Rosser property [8]. However, due to the undecidability of the algorithm, no confluence checking is actually performed. All graph grammars under RGG are assumed to be selection-free.

Since the confluence constraint imposes a restriction with an unknown impact on the language domain, an extension to RGG, called RGG+ [9], was developed. RGG+ generalizes the parsing process by sacrificing the confluence condition. However, it imposes a size-increasing condition on the production rules—the LHS size is less than or equal to the RHS size—resulting in a weak structural restriction on the grammar, while ensuring decidability [9]. It also reduces the LGG multi-layer decomposition of labels to the usual two layers of terminals and non-terminals. Due to the weak restriction, simplification, and a few low-cost preprocessing enhancements, RGG+ proposes that, while the worst case run time for parsing is exponential, the worst case is rarely encountered for many applications.

2.2 The Spatial Graph Grammar

The idea of a Spatial Graph Grammar (SGG) [9] developed from the RGG+ system. Many graphs contain inherent structural information such as relative relationships between nodes expressed as location, direction, topology, distance, etc. The spatial specification in SGG is generally employed as fuzzy logic categorizations such as north, south, east, and west, or near and far. The granularity and meaning of the categorizations is specified by the grammar designer. Using spatial specifications in the graph grammar imposes constraints on the search space during parsing. Those constraints improve the parsers runtime [9]. The amount of improvement depends entirely on the sparseness of spatial specifications embedded in the grammar. Grammars having no spatial specifications have a parsing run time equal to RGG+. Grammars with spatial specifications in every rule gain the most parsing runtime benefit.

As SGG represents the latest development for this line of systems, it represents a robust formalism to pair with a grammar induction system. However, the formalism is lacking in some areas—most notably with edge specifications. This area is addressed in the design of VEGGIE. The SGG system is also designed

for graph transformation as well as graph parsing. For graph parsing, the parser uses an R-application process—grammar rules applied right to left. For graph transformation, the parser uses an L-application process—grammar rules applied left to right. When using the grammar induction system, an R-application grammar is constructed for VEGGIE.

3. A graph grammar induction system

As interest in graph grammars has grown, so has the interest in applying machine learning principles to them. Several methods for inferring graph grammars have been developed since the early 1990s. Inferring graph grammars has significant research potential. The process can be used in data mining applications as inferred grammars contain frequently occurring substructures found in the graph dataset. A *substructure* is defined as an abstraction of some subgraph found in the dataset. An *instance* is defined as an occurrence of the substructure in the dataset.

These substructures represent hidden recurrent patterns within the larger graph. Substructures can also show divergent patterns between common recurrent patterns. Both help the researcher understand large, complex systems. Inference also aids in the construction of system generators. By inferring a graph grammar from existing graph data, the researcher may study the growth of complex systems. Using the grammar in a generator, the researcher can produce size constrained graphs that simulate a system's growth. Other applications are numerous.

For VEGGIE, a machine learning system for context-sensitive graph grammars is desired. The definition of context-sensitive graph grammars is that the left hand side (LHS) of production rules may contain one or more data elements—nodes of terminal or non-terminal designation. Context-free graph grammars are restricted to one and only one non-terminal on the LHS. The inference of context-sensitivity graph grammars is generally user centric. Finding contextual meaning within a system generally implies the analysis of attributes associated with data elements. While several context-free induction systems have been developed, only one context-sensitive induction system was found in literature [2] and its application is limited to a few well defined context-sensitive patterns.

Some popular graph grammar induction systems are AGM [10], FSG [11], and gSpan [12]. These three systems are essentially incremental enhancements to a common process. The latter systems provide improvements over the earlier. However, another system, SubdueGL [13], has evolved within its own uniform domain with little change to performance and its variants generally add optional functionality to the original system.

3.1 The SubdueGL induction system

The SubdueGL induction system uses a subgraph discovery approach that places emphasis on *compressing* the graph dataset as opposed to finding frequent subgraphs. While the compression and frequent subgraph approaches are closely related, they can produce different results as a less frequent subgraph may produce a better overall compression for the dataset. Using a subgraph growth process, SubdueGL generates candidate substructures that may be used to compress the graph dataset. The original method uses a minimum description length (MDL) compression value to compare each competing substructure's compression on the dataset. The substructure with the highest MDL value is used to compress the dataset. This process repeats until the dataset is either fully compressed—a single node remains or no substructure can be found—or a specified number of iterations is reached. The compression process results in an *hierarchical reduction* of connected subgraphs that directly corresponds to grammar production rules.

Since exact isomorphic subgraph discovery is NP-complete, an optional optimization uses an *inexact* isomorphic subgraph discovery based on a branch and bound algorithm to match subgraphs that vary slightly in their edges and nodes. Costs are associated with the variations. Substructure instances with the lowest cost within a threshold limit are selected for compression. When the threshold limit is set to zero, the inexact matching process reduces to the exact matching process.

The other systems use node adjacency matrices to maintain edge connections. SubdueGL maintains a list of nodes and edges. Each node contains an edge reference list and each edge contains its two node references. Therefore, it can handle graphs with multiple edges between two nodes and cycles.

Several additional options are noteworthy. Predefined subgraphs can be specified to compress the dataset before the normal induction process begins. It also performs as a supervised machine learner with the specification of a negative graph dataset. In addition, various options are provided to control the induction process, such as limits on the size and number of substructures to consider.

3.2 A SubdueGL derivative

The SubdueGL version used for VEGGIE provides both the MDL-based graph compression method and a sized-based alternative method [14, 1]. The size-based algorithm uses the same form as the

MDL algorithm to compute the final compression value. However, the size calculations for each portion of the algorithm are a simple sum of the node and edge counts. Therefore, its computation time is significantly less—O(1)—than the MDL calculation—$O(\log_2(n))$ average, $O(n\log_2(n))$ worst case. The result is a reduced runtime that occasionally selects a less optimal compression for the graph [14].

This version also introduces a change to the type of recursive substructures detected. The previous version allowed for a single *connecting* edge between substructure instances to define substructure recursion. The derived version replaced this edge recursion with node recursion. Node recursion allows for a single *overlapping* node between substructure instances to define substructure recursion.

4. The VEGGIE system

The VEGGIE system is essentially a combination of the SubdueGL and SGG systems with novel enhancements. Its user interface is primarily a refactored combination of SGG's three independent editors: the Node Type, Grammar, and Graph editors. Since SGG and SubdueGL are different systems, they each use data structures and processes optimized for their respective goals. Standardizing on a common framework is a goal for constructing VEGGIE. Several enhancements are made due to the standardization process. Other novel enhancements are introduced independently to each system.

As VEGGIE is a merging of the SubdueGL and SGG software, they require a standardization of terms, concepts, and structures. Since they both deal with graph data, VEGGIE endeavors to find a common ground based on a graph formalism. There are many formalisms for graph structures, but one formalism, GraphML, promises to become dominant in the standards community as many competing formalisms have been unified under its specification. GraphML is an XML specification for representing graphs of broad applicability and allows for general data extensions under the guise of XML attributes as well as a specifically defined "data" tag. As such, the SubdueGL and SGG applications are refactored using the GraphML specification.

4.1 Supporting GraphML

Much of the refactoring is a simple updating of terminology. The SubdueGL system referred to graph nodes as "vertices", but the SGG system referred to them as "nodes". As GraphML refers to them as "nodes", they are standardized as such in both systems. A "vertex" notation is used in SGG, but it refers to edge connecting endpoints within a node. These endpoints are called "ports" in GraphML and are standardized as such. GraphML also supports hyper-edges. Currently, VEGGIE does not recognize hyper-edges. As they are a grouping of standard edges, this could be implemented. This has been left for future work as there is little perceived gain.

Aside from terminology, VEGGIE implements several structural and visual changes. The GraphML formalism specifies data storage. The file types use the XML specification for GraphML. Data extensions as XML attributes are added to most GraphML elements. Data extensions as specified data types are added for any node type attributes and any production rule action code and spatial specifications.

4.2 The Type editor

The Type editor is refactored from SGG's Node Type editor. As an enhancement, edge types were added. In the display tree, two tree nodes appear in the editor's type tree to support the two different type lists: one for node types and the other for edge types. Node types contain a name, category, ports, and attributes. The categories were changed to the traditional terms, "Terminal" and "NonTerminal". Edge types contain a name and a directed attribute that specified whether the edge is directed or undirected from its source node to its target node.

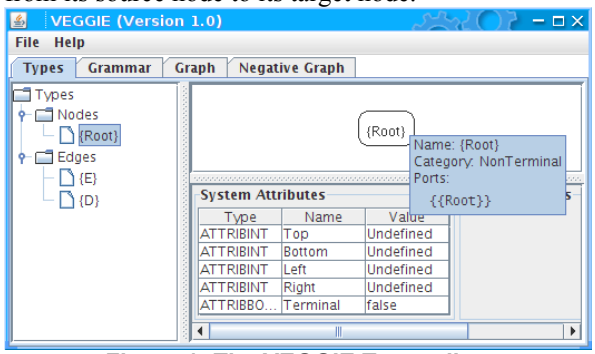

Figure 1: The VEGGIE Type editor

Another enhancement is default types. Default types cannot be added, renamed, or removed by users. All default names are denoted with surrounding brackets as users are restricted to starting a name identifier with alphanumeric characters. One default node type is generated with the name identifier "{Root}" and specified as a non-terminal. This default node type serves to specify the final node on the LHS of the final production rule. The parser recognizes the node with this default node type as the validating criteria when there is one and only one remaining node derived from parsing a graph dataset. For each node type, a default port is automatically generated with the identifier "{<NodeType>}" where "<NodeType>" is the name identifier for each node type. For example, the default port name for the "{Root}" node type is "{{Root}}". Two default edge

types are automatically generated in the edge type list. One is generated with the identifier "{E}" and specified as undirected. The other is generated with the identifier "{D}" and specified as directed. There are two purposes for maintaining default elements. The first is to allow users to create simple grammars and graphs from the default elements with a minimum of effort in the Type editor. The second, more important purpose, is to allow a less restrictive loading of graph data into the editors—defaults are used when edge data is not fully qualified:

1. For an edge, if a node port is not specified for a node, the default node port is used.
2. For an edge without a name, a default edge connects the nodes—the "{E}" default edge for unspecified or undirected edges and the "{D}" default edge for directed edges.
3. For a node, a default port with the given node type name is guaranteed. A node without a node type name is given the name "<BAD#>" and its default port is named "{<BAD#>}" where # is an enumeration for the currently malformed node.

Type specifications are stored separately from the grammars and graphs that use them. They are used to specify nodes and edges used in grammars and graphs. Therefore, a single type specification can be used for multiple grammars.

4.3 The Grammar editor

The refactored Grammar editor incorporates the node type changes and the edge type enhancements. In addition, the visual interface is updated in several important ways. The LHS and RHS display areas are essentially graph editors (see section 4.4 below). The graphs are related to represent a production rule. When nodes and edges are added, the user specifies its type from the currently loaded type specification. The action code and spatial specifications are supported as per SGG.

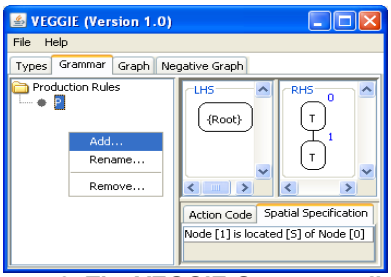

Figure 2: The VEGGIE Grammar editor

A grammar is a list of production rules. It is stored separately from the type specifications, graphs, and the parsing system. Therefore, a variety of grammars can be used to parse any given graph as long as the types used are contained in the current type specification.

4.4 The Graph editor

The refactored Graph editor allows for the adding and removing of nodes and edges. Node and edges are automatically given an integer identifier indicating an instance of their respective types. Node ports are used to connect edges and are selected during the edge adding process. The display area is also fully scrollable allowing for large visual representations of graphs.

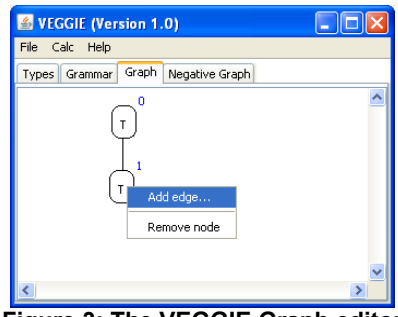

Figure 3: The VEGGIE Graph editor

In SGG, an invalid parse generally produced a malformed partial parse tree for display. As an enhancement, VEGGIE displays a corrected partial parse tree for review. The partial parse tree shows all the parsing done to the point of failure with all the proper links between production rules containing the nodes they used. This provides important feedback to the user for possible corrective actions or to identify aberrations within a graph.

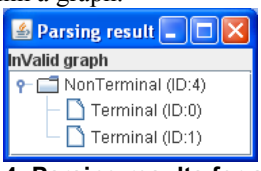

Figure 4: Parsing results for a graph

The addition of edge types has a significant impact on the parsing algorithm. Edges were previously considered undirected, type independent entities. Therefore, the parser represented the edge information as node adjacency lists with port connection information. Representing typed edges with directive information (directed or undirected) complicates the parsing process, but not significantly. Edge type and directive information provides constraint conditions that affect the parsing runtime. During the parser's search process, the port connection information is used to match connected production nodes to connected graph nodes. When additional edge information is provided, the matching process reduces the possible matching candidates as the search is constrained to a more limited edge set. The original edge matching code compared the end nodes' type and port connections between host graph and production rule. The modified code includes comparing the edge type and directive information. As the port

connection information is essentially an edge list for each port of a node, the edge directive information is coded to identify the edge as *undirected* between the nodes, *directed to the target* node, or *directed from the source* node.

Graphs are nodes and their connecting edges. Nodes are simple instances of node types. Edges are instances of edge types connecting two nodes via ports. A graph is stored separately from any type specification and grammar. Therefore, a graph may be related to a variety of grammars as long as the types used are contained in the current type specification.

4.5 Supporting SubdueGL

A refactored version of SubdueGL is integrated as a subsystem of VEGGIE. To support SubdueGL functionality, several changes and enhancements are incorporated. To support the supervised learning of SubdueGL via its negative graph function, an additional Negative Graph editor is added. The Negative Graph editor is identical to the Graph editor in every respect. Both graph editors have access to the same inference function that implements the SubdueGL subsystem. As a system validation check, the Graph editor must contain a nonempty graph while the Negative Graph editor is allowed to contain an empty graph for unsupervised learning.

To support the predefined substructures functionality of SubdueGL, production rules may be predefined in the Grammar editor. Any production rules that exist in the Grammar editor are interpreted as predefined substructures that compress the combined positive and negative graph dataset through R-applications. This allows the user to preprocess the graph data before the induction process begins.

VEGGIE supplies data to SubdueGL subsystem by translating the SGG graph dataset into a SubdueGL dataset. The work related to translation highlights the structural differences in each system. The SGG system uses node ports to connect edges while SubdueGL does not. Therefore, the SubdueGL subsystem is modified to accept and use ports. Most of these modification relate only to edge matching processes.

5. Context-sensitive induction

VEGGIE is intended to be a context-sensitive reinforcement machine learning system. As such, VEGGIE must be context-sensitive in both its parsing and inference processes. Context-sensitive parsing is provided by the original SGG system. However, SubdueGL is not context-sensitive. Previous work [1] has shown the usefulness of combining the two systems and proposed future work for context-sensitive induction. VEGGIE realizes these two objectives by combining the systems and extending the SubdueGL subsystem with novel work for context-sensitive induction.

Useful context-sensitive induction could include user provided contextual information to direct the induction process. The contextual information could target a particular contextual domain such as a spatial domain constrained to the major two dimensional vector directions as specified by the SGG formalism (N, S, E, W, NE, NW, SE, SW), an expanded three dimensional formalism, or some other *fuzzy* spatial context. However, the SGG system allows for the specification of general context-sensitivity without formal domain constrains. Therefore, the novel work extends the induction process in VEGGIE to include general context-sensitive induction.

5.1. The context-sensitive inference problem

Context-sensitivity in graph grammars is defined as allowing the LHS of a production rule to contain one or more non-terminals and zero or more terminals. To ensure a halting condition during parsing, a size-increasing constraint on the production rules can be imposed as per RGG+[9]. To ensure proper node assignments (non-terminals, terminals, and their edge connections), the rules must provide node context and port marking information describing the embedding relationship between LHS and RHS nodes. In the SGG system, a production rule's LHS and RHS respective nodes are given contextual markers that inform the parser about node and edge reassignments. Ports are given contextual markers that inform the parser about edge connection reassignment. These issues impose problems for an induction process that is required to infer context-sensitive production rules.

To create a context-sensitive production rule, the induction process must identify substructures within the host graph that cannot be simply reduced to a single non-terminal as is done for context-free grammars. These substructures must contain some property that challenges the induction of a context-free production rule. To ensure the halting condition, an inferred context-sensitive production rule must reduce two or more RHS nodes to a single LHS non-terminal node. This ensures that the LHS size is less than the RHS size—a requirement of the halting condition.

Non-terminal nodes are generated during the induction process. As they are not predefined node types, they are added by the induction system. These non-terminals are free of multiple port definitions as a context-free induction system only relies on the default ports. However, for context sensitive rules, marking plays an important role in replacing RHS substructure in a host graph with LHS substructures—

the *embedding* issue. Marking is used on ports and nodes in the definition of production rules to maintain edge connections outside an instance in the remaining host graph.

5.2. Inferring context-sensitive grammars

Previous work for inferring context-sensitive graph grammars [2] was limited to matching general predefined patterns within the host graph. These general patterns consisted of chain and star patterns containing nodes and directed edges of general type. As chain patterns of size *M* and star patterns of size *N* can be limited, the method highlights pattern growth through the recursive use of production rules.

However, the work did not address pattern overlap. Two patterns may overlap by sharing common nodes and edges within the host graph. The chain and star productions were limited to unconnected nodes on the LHS, therefore, patterns only overlapped on nodes. Furthermore, only end nodes could modify chains to produce longer chains and stars only added edges between a central node to the other nodes. Therefore, overlap was a trivial chain extension problem for chains and a trivial chain connecting problem for stars.

Within SubdueGL, overlapping instances of discovered substructures can be identified when the *instance overlap* parameter is used. Overlapping instances are then considered non-overlapping and compressed as any other instance. This is unsuitable for generating a general context-free production rule as the parsing system fails on the induction graph since individual instances are identified, but they are ignored as recursive instances sharing a common overlapping structure.

To handle basic context-free overlapping instances, a *node recursion* parameter is used to allow instances sharing a single node to specify node recursive production rules. An *edge recursion* parameter is also allowed, but the instances are not considered overlapping as the edge is not shared by the instances—it connects the instances. For node recursion, there is only one node shared between multiple instances, so there is no requirement to provide embedding information. A single non-terminal with its default port suffices to replace each instance's overlapping node as well as an entire instance. The result is a context-free recursive production rule. However, embedding information can be applied to the production rule by copying the ports of the overlap node to the new non-terminal node and marking all ports between the overlap node and its non-terminal.

Expanding the instance overlap and single node recursion provides a mechanism to infer context-sensitive production rules. When the overlap between instances is a substructure consisting of one or more nodes, each of the overlap nodes form new non-terminal nodes with identical ports that require marking to resolve their embedding requirements in the host graph. Consider the example graph in figure 5. The dotted oval shows the overlap of three instances of a common substructure.

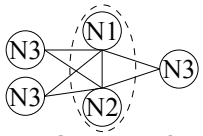

Figure 5: An example graph

The resulting recursive production rule could be as shown in figure 6. Embedding marks are not shown in this simple example.

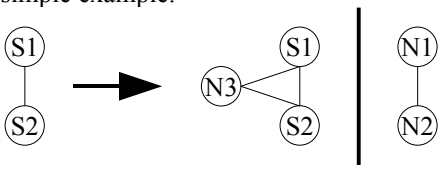

Figure 6: Two context-sensitive production rules

It is noteworthy to mention that a context-free solution could be applied. The overlap could be reduced to a single non-terminal node before a recursive rule is constructed. However, overlapping structures embody a contextual reference between common instances. Context can be lost using this approach as embedding information between the overlap and non-overlap areas is undefined. Therefore, the reduction process cannot preserve structural context within the recursive instances. A context-sensitive solution is required.

The process for constructing a context-sensitive solution relies on the mapping of instances to their shared overlapping substructure. As multiple instances may overlap, the system must ensure that the shared substructure applies to all concerned instances. The mapping provides the required embedding information to mark ports for context preservation.

Chains of overlapping instances are also possible and must be captured in the resulting production rules. In this case, overlapping and non-overlapping areas can be reinterpreted and exchanged producing a "chicken-or-egg" problem when constructing the rule. A solution is to let both versions compete for the best compression value with ties broken arbitrarily.

6. Conclusion and Future Work

VEGGIE provides a context-sensitive parsing and induction environment—the first of its kind for the visual computing domain. SGG provides a graph formalism and parsing system. SubdueGL provides a graph grammar induction system. GraphML provides a comprehensive data format and file structure. As a reinforcement machine learning environment,

VEGGIE extends artificial intelligence in the graph grammar domain. The graph grammar induction process provides a mechanism for quickly constructing visual languages from example graphs.

Context-sensitive induction is extended from the context-free induction of overlapping single node recursive production rules. Overlap between instances of *common* substructures provides a connection context between those instances. Mapping the substructure defined by the overlap to the instances provides the means for extracting embedding information used in a production rule.

As VEGGIE is a new tool, few tests on real world examples have been applied. Comprehensive tests are planned for data related to bioinformatics, social networks, and adaptive web interfaces [3]. However, some initial evaluation of VEGGIE has been performed. Tests were conducted on the examples provided in previous work [1] and previous SubdueGL examples. The grammar induction tests validate the grammar construction process. The grammar parsing tests were equal to the results of the previous work's tests. Accuracy of the induced grammars was preserved when the parser was used against representative graphs of the grammars.

A future goal of VEGGIE is to find context between different substructures. Overlapping instances of *differing* substructures also provides an opportunity for connection context. Finding these overlaps are problematic as this assumes the substructures have previously been inferred from the graph dataset. As each inferred substructure affects later inference processing, a different approach to grammar induction may be required.

SubdueGL and SGG both have a isomorphic subgraph matching function. SubdueGL uses its function to find substructures in the dataset to apply compression. SGG uses its function to find grammar instances in the dataset for parsing resolution. As both functions share common applicability, they should be code compatible. However, they both take different approaches to matching subgraphs. A future goal of VEGGIE is to reduce both of these functions into a single process.

7. Acknowledgments

Thanks goes to Dr. Larry Holder for SubdueGL assistance and Dr. Jun Kong for SGG assistance.

The VEGGIE project was funded in part by the GetDoc program from the University of Texas at Dallas and the Graduate Assistance in Areas of National Need (GAANN) program from the U. S. Department of Education.

8. References

[1] Ates, K., Kukluk, J., Holder, L., Cook, D., and Zhang, K., "Graph Grammar Inference on Structural Data for Visual Programming", *Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2006)*, 2006

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

[3] Zhang, K., *Visual Languages and Applications*. Springer, New York, NY, pp. xiii, 2007

[4] Chang, S. K., "Picture Processing Grammar and Its Applications". *Information Sciences* 3:121-148, 1971

[5] Smith, D. C., "Pygmalion: A Computer Program to Model and Simulate Creative Thought". *Ph.D. Thesis*, Stanford University, 1975

[6] Sutherland, I. E., "Sketchpad: A Man-machine Graphical Communications System". *Ph.D. Thesis*, MIT, 1963

[7] J. Rekers and A. Schurr, "Defining and Parsing Visual Languages with Layered Graph Grammars". *Journal of Visual Languages and Computing*, 8(1):27-55, 1997

[8] D. Zhang, K. Zhang, and J. Cao, "A Context-sensitive Graph Grammar Formalism for the Specification of Visual Languages", *The Computer Journal*, 44(3):186-200, 2001

[9] Kong. J., "Visual Programming Languages and Applications". *Ph.D. Thesis*, Department of Computer Science, The University of Texas at Dallas, 2005

[10] A. Inokuchi, T. Washio and H. Motoda, "An Apriori-based Algorithm for Mining Frequent Substructures from Graph Data", *Proceedings of the European Conference on Principles and Practice of Knowledge Discovery in Databases*, 2000

[11] M. Kuramochi and G. Karypis, "An Efficient Algorithm for Discovering Frequent Subgraphs", *Technical Report 02-026*, Department of Computer Science, University of Minnesota, 2002

[12] X. Yan and J. Han, "gSpan: Graph-Based Substructure Pattern Mining", *Proceedings of the International Conference on Data Mining (ICDM)*, 2002

[13] I. Jonyer, "Context-Free Graph Grammar Induction Based on the Minimum Description Length Principle," *Doctoral Dissertation*, The University of Texas at Arlington, August 2003

[14] J. Kukluk., L. Holder, and D. Cook, "Inference of Node Replacement Recursive Graph Grammars", *Sixth SIAM International Conference on Data Mining*, 2006