Learning Geometric Graph Grammars

Marek Fiser Bedrich Benes Jorge Garcia Galicia Michel Abdul-Massih Daniel G. Aliaga Vojtech Krs

Purdue University

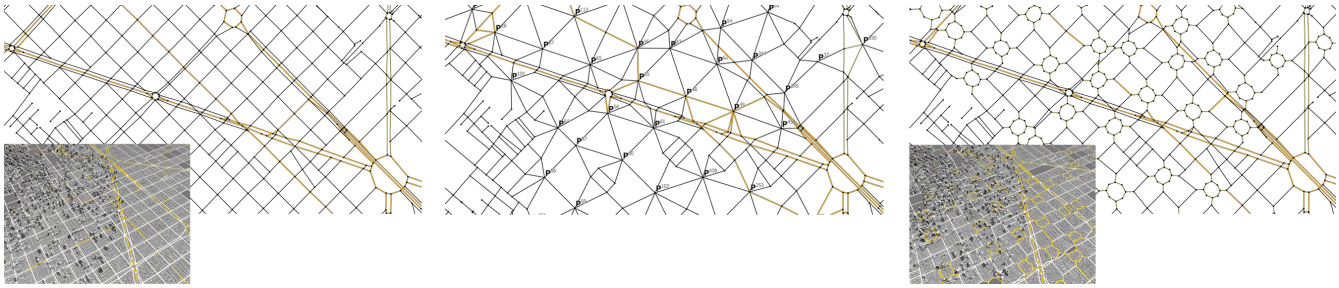

Figure 1: *A geometric graph grammar was learnt from the road network of the city of Barcelona (left) and similar structures were replaced by a single node (middle). Detected similar subgraphs were encoded as terminal symbols of the grammar and replaced by a manually selected subgraph (right).*

Abstract

We introduce geometric graph grammars, demonstrate how they can generate geometric structures, and introduce an algorithm for their automatic learning (inverse procedural modeling). Our approach extends the concept of graph grammars to allow for coding not only topological data, but also geometry. Forward modeling generates geometric graphs and considers various strategies for node connectivity. Inverse procedural modeling performs learning of geometric graphs, by discovering repeated structures and their connectivity. These structures are encoded into geometric graph grammar rewriting rules. We demonstrate usability of our approach on an example using urban networks. Graph learning is reasonably fast; in our implementation, learning of a road network with $72k$ vertices and $100k$ edges is performed in less than one minute.

Keywords: Graph Grammars,Inverse Procedural Modeling, Shape Representation, Grammar Learning, Computer Graphics

Concepts: •**Computing methodologies** → **Shape analysis;** *Mesh models; Parametric curve and surface models;*

1 Introduction

Capturing, modeling, and representing real-world structures are important outstanding problems in computer graphics. One popular representation is procedural modeling that expresses structures in a compact manner. However, the definition of procedural models is complex and precludes their wider use in real-world applications. Graphs are common in applications that require information about topological connectivity and graph grammars have been used successfully for a long time. Surprisingly, graphs that include geometric information are not especially common in grammar representations.

We extend graph grammars so they consider geometric information, thus merging the expressive power of procedural models with graph algorithms. This extension is non-trivial. Since a graph grammar can replace a single node with a sub-graph, an algorithm is needed to define the connectivity of the inserted sub-graph. An additional problem is the creation of a procedural representation of an existing graph - a task called inverse procedural modeling. Similar to dictionary-based encoding, large and/or frequently repeated blocks are encoded as terminal symbols of a grammar. Higher-level structural blocks and their connections are encoded as non-terminal symbols and rules.

We introduce context-free non-parametric geometric graph grammars that extend graph grammars such that they allow for coding not only topological data, but also geometry. Inspired by L-systems [Prusinkiewicz 1986], we have generalized the concept of turtle interpretation of the string of modules. L-systems, however, are linear, while our approach allows for graph rewriting. We also present a new algorithm for learning geometric graphs.

We demonstrate our approach on practical examples of urban road network learning and synthesis. We show how frequently repeated structures can be detected and a new graph can be synthesized or an existing graph can be modified by reusing the learnt geometric graph grammar and its rules. Our work claims two main contributions: 1) the definition of a geometric graph grammar that allows for encoding topological and geometrical information and, 2) an algorithm for learning geometric graph grammars (inverse procedural modeling).

2 Previous work

Graph grammars (graph rewriting or graph transformation) study how to generate one graph from another graph. In the context of computer graphics, graph grammars have had limited use in procedural modeling [Siromoney and Siromoney 1987] and our work is based on the Node Label Controlled Graph Grammars (NLC-GGs) that were introduced in [Janssens and Rozenberg 1980] and extended in [Engelfriet and Rozenberg 1991].

Graph inference, or graph learning, techniques are used to induce

SCCG'16,, April 27-29, 2016, Smolenice, Slovakia
ISBN: 978-1-4503-4436-4/16/04
DOI: http://dx.doi.org/10.1145/2948628.2948635

grammars from input graphs [Nevill-Manning et al. 1994; de la Higuera 2010]. Graph grammar induction is a method by which a set of graph grammar production rules are obtained by analyzing a given graph [Jiang et al. 2012]. A key component of graph learning is the study of algorithms that find frequent subgraphs contained in a graph [Cook and Holder 2006]. One of the fundamental inexact frequent graph learning algorithms is SUBDUE by [Cook and Holder 1994], which uses the minimal description length principle for substructure discovery inside graphs. Recently, Blockeel and Nijssen [Blockeel and Nijssen 2008] extended SUBDUE to develop algorithms on NLC-GG induction. Our method is inspired by Blockeel and Nijssen [Blockeel and Nijssen 2008], but it is different in two main ways. First, we are working with geometric and topologic graphs whereas [Blockeel and Nijssen 2008] deal with purely topological graphs. Second, they assume that the input graph is already labeled, but in our work the grammar is not label controlled. Kuramochi and Karypis [Kuramochi and Karypis 2007] describe gFSG, an algorithm for finding frequent geometric graphs invariant to rotation, translation or scaling. Our approach also finds subgraphs that are invariant under such transformations, but we use vertex expansion to robustly find all subgraphs in large inputs.

Early work on inverse procedural modeling addressed automatic detection of repetitive urban structures in [Aliaga et al. 2007]. The work of [Šťava et al. 2010] detects an L-system for an input vector image. This work, however, detects linear structures and does not provide a procedural model for complex topologies. Bokeloh et al. [Bokeloh et al. 2010] described a connection between triangles and an underlying procedural model for 3D meshes. There are various methods that use an existing procedural model and attempt to find an optimal set of parameters. Talton et al. [Talton et al. 2011] use Metropolis optimization to fit a 3D procedural model into predefined constraints. Vanegas et al. [Vanegas et al. 2012] used Metropolis-Hasting optimization to learn a set of input parameters for a procedural model that generates a city model with given properties. Talton et al. [Talton et al. 2012] induced a probabilistic grammar over a set of design patterns using Bayesian networks and Stava et al. showed how to find a procedural representation for biological trees [Šťava et al. 2014]. Most of these approaches depend upon an existing procedural model and attempt to find parameters for a given output that would generate this model.

An important field that deals with the topic of encoding geometry into grammars is pattern recognition. Han et al. [Han and Zhu 2005] reconstruct a scene of mainly rectangular shapes using attributed graph grammars. Lin et al. [Lin et al. 2009] use attributed stochastic graph grammars for generation and recognition of categories. Schmittwilken et al. [Schmittwilken et al. 2009] demonstrate how to construct 3D buildings using attributed graph grammars. We refer the reader to the book [Zhu and Mumford 2007] for an overview of recent algorithms in this field.

3 Geometric Graph Grammars

We introduce geometric graph grammars (GGGs) by extending node label controlled graph grammars [Engelfriet and Rozenberg 1991; Blockeel and Brijder 2009] and by generalizing the concept of turtle interpretation of L-systems [Prusinkiewicz 1986].

3.1 Geometric Graph Definition

A *graph* is an ordered pair $G = (V, E)$ of a non empty set of vertices V and a set of edges $E = \{e = \{a, b\} | a, b \in V, a \neq b\}$, such that each edge $e \in E$ connects two vertices from V. A graph is said to be *simple* if it is unweighted, undirected, without loops, and without parallel edges. A graph is a *geometric graph* if every $v \in V$ has a spatial position assigned so a vertex v has two coordinates $c(v) = (x, y)$ in 2D.

We say two graphs G and H are geometrically isomorphic when there exists a bijection $f : V_G \rightarrow V_H$, such that $\{a, b\} \in E_G$ iff $\{f(a), f(b)\} \in E_H$ and there exists a transformation $T : R^2 \rightarrow R^2$, such that $T(c(a)) = c(f(a)) \forall a \in V_G$, i.e., coordinates of corresponding vertices do not change after transformation. Although T could be any transformation, we use affine transformations.

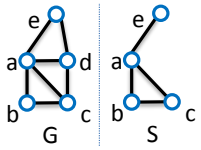

Figure 2: *Induced subgraph S of G.*

Given a set of graphs $X = \{G_1, G_2, \ldots, G_n\}$, we call call X an *isogroup* if $G_1, G_2, \ldots, G_n$ are geometrically isomorphic to each other.

A graph $S = (V_S, E_S)$ is a subgraph of G if $V_S \subseteq V$ and $E_S \subseteq E$. A subgraph $S \subseteq G$ is called induced iff E_S consists of all edges in E_G that have both endpoints in V_S. Figure 2 shows a graph G and an induced subgraph S; S contains vertices a, b, c, e from G and must include all edges that connect these vertices in G and no additional edges.

3.2 Geometric Graph Grammar Definition

We define a context-free deterministic non-parametric geometric graph grammar (GGG) that replaces a single node with a graph. A GGG *production* (or rewriting) *rule* has the form (see Figure 3)

$$q \rightarrow S/B, \tag{1}$$

where q is the replaced node, S is the subgraph that replaces q in the main graph G, and B is the geometric graph embedding. The geometric embedding has the form

$$\begin{aligned} B \quad = \quad & \{(v_0, T_0, \{p_{0_0}, p_{0_1}, \ldots\}) \\ & (v_1, T_1, \{p_{1_0}, p_{1_1}, \ldots\}), \ldots \\ & (v_n, T_n, \{p_{n_0}, p_{n_1}, \ldots\})\}, \end{aligned} \tag{2}$$

where $i = 1, \ldots, n$ is the number of vertices, $v_i \in S$ is a vertex, T_i is a transformation of the local coordinate system of v_i, and $p_{i_j} = [x_{i_j}, y_{i_j}]$ specifies coordinates of the embedding vertices relative to v_i i.e., vertices that are supposed to be present in the other graph and that will connect to v_i.

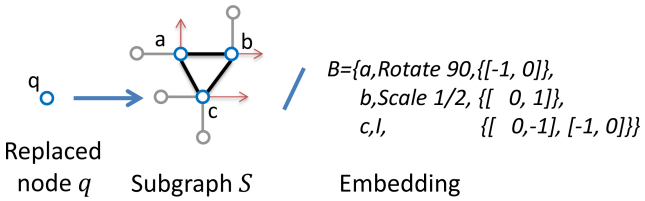

Figure 3: *A GGG production rule that replaces a node q with a graph and its embedding. The gray edges and circles denote the expected vertices in the graph where the subgraph will be connected, the red arrows symbolize one of the axes of the transformed local coordinate system used for next rule placement.*

Linear string rewriting systems, such as L-systems, encode the geometry explicitly in the symbol parameters or implicitly using a transformation table (e.g., " + " means "turn right"). The geometry is produced *after* the list of symbols has been generated from the production rules locally by the turtle [Prusinkiewicz 1986]. In

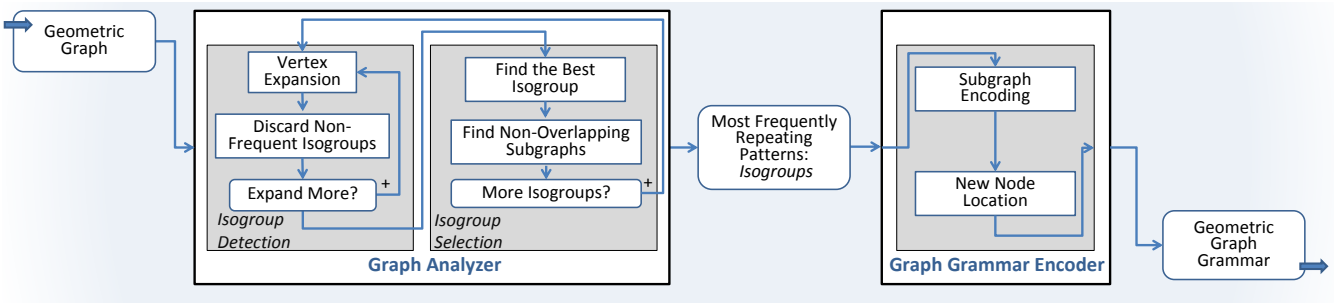

Figure 4: *The geometric graph analyzer takes the input graph and finds the list of the similar subgraphs called isogroups. Isogroups are encoded as a set of rules by the graph grammar encoder.*

GGG the geometry changes affect the graph globally so it would be difficult to use post processing. Instead, we generate the topology and the geometry in a single pass. In GGG each vertex stores the associated transformation T_i that defines its local coordinate system and the initial transformation must also be provided explicitly in the axiom ω, i.e., in the starting symbol of the rewriting.

A GGG production rule is shown in an example in Figure 3. It replaces a node q with a graph

$$S = (\{a, b, c\}, \{(a, b), (b, c), (c, a)\}),$$

with embedding

$$\begin{array}{lll} \{(a, & Rotate\ 90^o, & \{[-1, 0]\} \quad), \\ (b, & Scale\ 1/2, & \{[0, 1]\} \quad), \\ (c, & I, & \{[0, -1], [-1, 0]\} \quad)\}, \end{array}$$

where I is the identity matrix. The empty nodes originating in a, b, and c are the embedding connections to the vertices that are anticipated in the main graph G, which is the graph on the left side of the production, and the arrows represent the coordinate system after transformation T_i. More specifically, node a has an embedding of a vertex at location $[-1, 0]$ as indicated by the gray node. The red arrow pointing up from the node a indicates how a new rule in a new derivation, if applied, will be oriented with respect to node a. Note that node c has two embedding edges.

Rule Application occurs in four steps. 1) The rule is transformed into the local coordinate system of $q_i \in G$ according to the transformation T_i. 2) Vertex $q_i \in G$ is erased, together with its connecting edges. 3) S is placed relative to q_i inside G. 4) The embedding B is connected to G; i.e., the vertices of the instance of S_i are connected to those vertices in G that are at the location $p_i = [x_i, y_i]$ indicated by the embedding B_i.

3.3 Relaxed Embedding

The embedding imposes a strong restriction on rewriting. If a vertex cannot embed its entire subgraph, it cannot be replaced. Designing such rewriting is difficult and impractical, although possible as shown later in Section 7. In our work, instead of requiring strict connectivity where each node of the subgraph needs to embed all vertices, rewriting is always allowed even with partial connectivity between the newly inserted nodes and the surrounding existing nodes.

Figure 5 shows such relaxed embedding. Node d in Figure 5 i) is to be replaced by the subgraph from Figure 5 ii). The resulting graph depends on the local coordinate system of d, three examples of these systems are illustrated by the red arrows. If the identity transformation is applied, the embedding does not attach to any node because the distance to other nodes is larger than a system imposed parameter. The production rule is still executed (Figure 5 iii)). However, if the rule is rotated by $\pm 45^o$, the embedding connects to vertices b or a, c respectively. Note that in Figure 5 iv), only one vertex b is embedded.

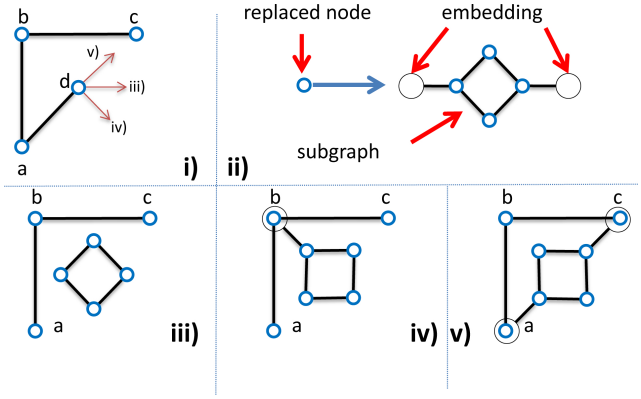

Figure 5: *Relaxed embedding. The node d from the graph in i) is replaced by the subgraph from the production ii) If the structure is rotated, the embedding might not connect to anything, as in iii) or either to vertex b in iv) or to vertices a, c in v) .*

4 GGG Learning Overview

Figure 4 presents a pipeline summary of the GGG learning. Given a 2D geometric graph, our method analyzes and encodes it into a set of GGG rewriting rules.

The Graph Analyzer finds a set of isogroups (groups of isomorphic graphs) in the input. Each isogroup represents a particular subgraph of the input graph that repeats frequently. The isogroups are built bottom-up. The algorithm starts with a single edge and forms a single isogroup. The *Isogroup Detection* iteratively expands each isogroup by one node until a maximum graph size is reached. The *Isogroup Selection* block of the graph analyzer takes the set of isogroups and selects the largest isogroup that satisfies additional constraints, such as being overlap-free. The data analyzer then repeats the process until no further isogroups can be detected. The result of the graph analyzer is a set of the most frequently repeated subgraphs. While our automatic search starts with a single edge, the user can also initiate the search with a user-defined graph. This *user-assisted learning* enables searching for user-defined subgraphs.

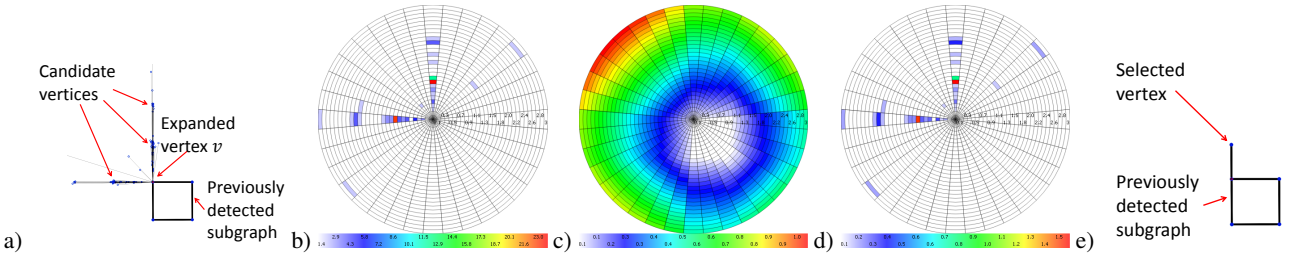

Figure 6: *a) Candidate vertices for expansion are counted in a discrete grid shown in b). The weighted contribution to the newly added area of the graph is shown in c), and the actual value of the area multiplied by the frequency of repetition is shown in d). The expanded graph by the selected vertex is shown in e).*

The Graph Grammar Encoder creates rewriting rules that replace each occurrence of an isogroup's subgraph by a single graph vertex. The subgraph is erased and the new vertex is embedded by connecting it with its neighbors. At the same time a rewriting rule is generated. The new graph is then encoded again generating a higher hierarchy level.

5 Graph Analyzer

Inspired by data compression, we seek to find a grammar that efficiently codifies large, or frequently repeated structures of a graph and the graph analyzer finds the most frequently repeating patterns.

5.1 Isogroup Detection

Isogroup detection attempts to find all isogroups of the most frequently repeating subgraphs by iteratively expanding previously found isogroups. The algorithm is initialized with a seed graph F that is in the simplest case a single edge. Alternatively, in the case of the *user-assisted learning*, the F is given by the user by selecting a part of the graph. The seed graph is iteratively expanded by adding a vertex. The vertex selection makes sure that the expanded graph is frequently repeated in the input graph. As the graph becomes larger, the likelihood of multiple occurrences decreases.

An example from Figure 7 that shows the analysis of the graph from the upper right-hand corner. The isogroup detection algorithm proceeds in a loop that tests the neighboring vertices of each isogroup in each iteration. The algorithm starts with a single edge and forms a single isogroup that contains all edges from the graph (first row of Order 2 in Figure 7). Both vertices are expanded resulting in graphs of Order 3 (we only show a sample of the candidates). The resulting graphs have three vertices and their number of repetitions is given by the multiplicity of the added vertex. The most frequent groups are then expanded further to Order 4 by adding a new vertex. The algorithm continues until no more structures can be detected. In each iteration the algorithm provides all subgraphs of the given order and their repetitions. Note that the same graph can be found as a result of expansion from different directions. These graphs are merged as indicated in the circular inset.

The **vertex expansion** takes an existing graph and extends it by one vertex such that the new graph is frequently repeated in the input. Vertex expansion proceeds by iteratively adding a new vertex q. Assuming that we have a frequently repeated graph F, vertices from all occurrences of F in the input are analyzed and the new vertex q is selected by using two heuristics: (1) the new graph is the most frequently repeated in the input, and (2) the new vertex increases the area of the new graph (the area of the convex hull). The first heuristic finds frequently repeated graphs, and the second one prevents the algorithm from finding long edges and generates results

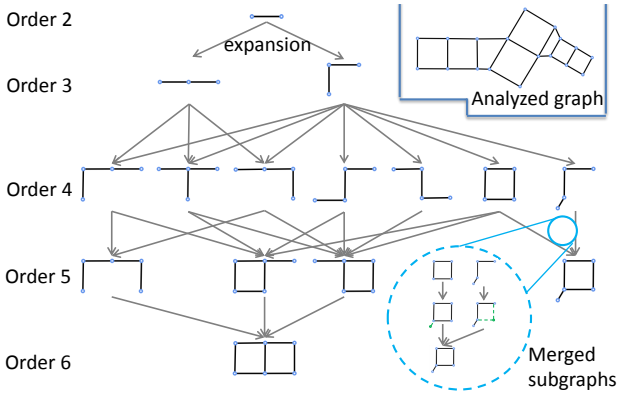

Figure 7: *Generation of the frequently repeated large subgraphs. The algorithm expands edges from the level* Order 2 *to all possible cases resulting in* Order 3*, then detects isomorphic graphs and prunes them. The algorithm continues by expanding* Order 3 *to* 4*, etc.*

that prefer bifurcations.

For each vertex we need to test many possible expansions and we simplify this step by using two discrete grids in polar coordinates. One grid is a counter of candidate vertices $f(\alpha, d)$ denoted by $\Delta a(\alpha, d)$ (Figure 6 b)), and the other counts the added area (Figure 6 c)). The expanded graph is transformed into a canonical normalized form with the expanded vertex in the origin of the coordinate system (labeled v in Figure 6 a)). Normalization is done such that all expanded graphs are aligned to each other (same scale and rotation). We then traverse all occurrences of the graph and test all expansion vertices by adding them to their corresponding cells in both grids. After all vertices have been visited, the algorithm calculates a score for each cell by multiplying the frequency and the area (Figure 6 d)). The cell with the highest value is selected. The new vertex q is positioned on the center of the cell and a new edge connecting the expanded vertex q to F is added (Figure 6 e)).

The discretization of the polar coordinate provides quantization of the vertex neighborhood, helps to avoid difficulties with numerical precision, and increases the speed of evaluation. The resolution of the grid affects the number of detected subgraphs and the level of their similarity. As shown in Figure 8, increasing the grid cell size (i.e., decreasing the grid resolution) detects fewer subgraphs but increases their similarity. The actual resolution depends on the input graph and we use discretization into 10×12 cells in all examples in this paper except Figure 6, where we demonstrate the smooth transition between different cells. We discuss the effect of the grid

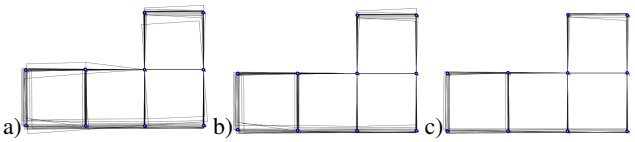

Figure 8: *The grid resolution (Figure 6) affects the precision and the count of detected subgraphs. By increasing the resolution we detect fewer, but more similar subgraphs. For this example, the resolution a)* 6×8 *detects* 15 *subgraphs, b)* 10×12 *finds* 13*, and c)* 20×24 *detects* 9 *subgraphs.*

resolution further in Section 7 in Figure 11.

Accelerating Isogroup Search Since the number of generated isogroups by vertex expansion can be large, we exploit two observations to accelerate performance. First, not all groups are important for further analysis. Certain expansions produce graphs with a low number of repetitions and not expanding them allows for a speedup of the search. Second, the same isogroup can be detected by expanding different subgraphs.

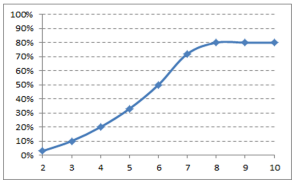

Figure 9: *The percentage of removed isogroups as a function of the iteration.*

Frequency-based Removal. The isogroups are sorted by the number of repetitions with a certain percentage of the last isogroups discarded. The cut value is smaller at the beginning of the analysis when the isogroups are not well-formed (Figure 9). The actual threshold is close to 90% but we use conservative 80% because there was no significant performance hit. We provide further results and comparisons in Section 7.

Isomorphism-based Removal. Since we work with induced subgraphs, it is possible to detect the same isogroup from different directions of the search. Figure 7 shows an example of detection of the isogroup of Order 5 in column B from different patterns of Order 4, as shown in the circular inset. This is why we merge isomorphic graphs after each vertex expansion. However, testing graph isomorphism is slow due to the need to check the topological and the geometric isomorphism (see Section 3.2), we perform this step after frequency-based isogroup removal.

The result of this step is a frequency table that stores the isogroups, their repetition count, their embedding, and the actual locations in the input graph. The detected geometric subgraphs can be *overlapping*.

5.2 Isogroup Selection

The size of the isogroup (i.e., the order of the graph) is controlled either explicitly by the number of performed iterations, or implicitly by stopping the algorithm, when no further expansion is possible. In most examples in this paper the complete analysis of the input was applied.

The largest from the most-frequently-repeated isogroups is selected. We can impose additional geometric constraints such as isogroups with near-to-one aspect ratio or graphs with large number of leaves as shown later in Section 7 in Figure 18. All subgraphs of G that belong to that isogroup are found, and their largest non-overlapping subset is detected. To do this, we create a dual topological graph that stores each graph as a node and codes two non-overlapping graphs by an edge that connects them. We then find the largest clique of the dual graph, which gives us the biggest non-overlapping subset of graphs within the isogroup.

The graph analyzer proceeds until there is no other isogroup available. The output of this step is a list of isogroups that are sorted by the order of the most frequently present.

6 Graph Grammar Encoder

The graph grammar encoder converts a set of isogroups into geometric graph grammar rules. The input of the encoder (Figure 4) is an ordered list of selected isogroups. The encoder's output is a group of rules with an embedding.

Right-hand Side Rule Generation The isogroup contains the subgraph S, its occurrences in the input graph, and its embedding B_i Eqn.(2). The subgraph S is unique, but its embedding and transformations can vary for each instance of the rule. This information is simply copied to the rules.

Left-hand Side Generation The left-hand side of the rule is a new node that replaces the graph. As we process the geometric graphs we need to determine the location of the newly created node. The straightforward solution is to place the node in the center of the replaced subgraph as shown in Figure 10 b). This causes star-like artifacts that change the graph appearance and cause decoding difficulties in later iterations.

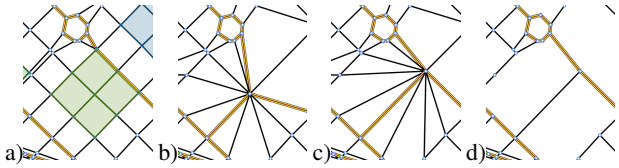

Figure 10: *The highlighted subgraph in a) is replaced by a single node. If the new node is located in the center of the subgraph b) it causes distortion of the original graph. A better solution is to locate the new node at the position of an existing vertex from the original subgraph, but the connecting edges can be visually distracting c). This can be alleviated by not rendering some edges d).*

The location of the new node depends on the application area, for example if we encode an urban graph, or a 2D representation of art, and is determined by considering properties of the input. In the case of road networks, we place the new node close to the streets with higher importance (arterials, highways), (Figure 10 c)) which minimizes the angles between the original graph and the embedding edges. The new node embedding stores the connection to all nodes from the original subgraph (p_{ij} in Eqn.(2)), so we do not need to keep all the edges when substituting a subgraph with a single node as shown in Figure 10 d).

Hierarchical Encoding Once subgraphs are replaced by the nodes, learning can continue with higher-level rule generation.

7 Results

Our system was implemented in C++, used BGL Boost for graph representation, Qt for direct rendering, and Open Street Map files as input. Our implementation was optimized for multi-core computing and all tests were run on eight cores of an Intel i7 CPU clocked at 3.2 GHz equipped with 16 GB of memory.

Forward GGG application An example in Figure 12 shows the production of a recursive Sirepińsky triangle using GGG. The axiom is a triangle with nodes labeled P. Each vertex in the axiom

Figure 11: *Effect of the grid resolution for vertex expansion on the decoded graph. The original model a) was encoded into three levels of hierarchy using grid resolution* 10×12 *cells and decoded back b). If higher resolution of* 20×24 *is used, the result provides better resemblance to the original c).*

has a local coordinate system oriented in the direction of the edge that connects the node with its neighbor in an anticlockwise direction (i.e., the red arrow in Figure 12). The first rule replaces vertex P with itself and the additional structure forming the rest of each triangle. Each transformation in the non-terminal symbol scales the structure by half and rotates by 0^o, 120^o, and 240^o as indicated by the arrows in the rules. The second rule replaces each vertex H with a more complex structure. All rules are applied in parallel.

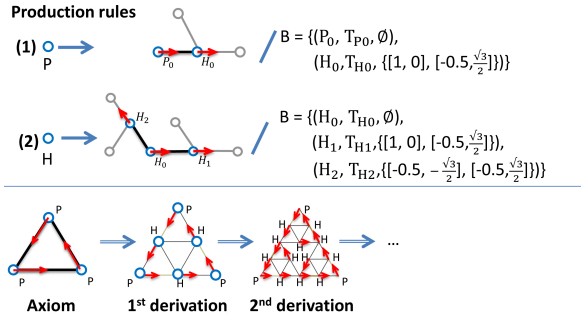

Figure 12: *Geometric Graph Grammars can produce recursive structures.*

Inverse GGG Modeling The graph grammar generation in Figure 13 shows several iterations of a complete encoding of Paris into a single node. The fully automatic analysis took 12 minutes and it was generated in 37 iterations (i.e., the rules have 37 levels of hierarchy). The total number of rules is $3,098$.

Vertex Expansion and Grid Resolution The effect of grid resolution on the vertex expansion algorithm is shown in Figure 11. Part of the map of Paris in Figure 11 a) was encoded into three levels of hierarchy and then decoded back. The reconstructed graph has significant visual artifacts for a vertex expansion grid of resolution 10×12 as seen in Figure 11 b). The errors are diminished if a higher resolution of 20×24 cells is used as in Figure 11 c).

Noise Sensitivity Figure 14 shows the result of a detailed analysis of the robustness of the algorithm against noise. The input was a regular mesh and the parameters of the analysis were set to detect the smallest rectangular graphs of size four. As shown in the first column, the algorithm found 100% of them. We have injected noise into the graph by jittering. The percentage of jittering in each column refers to the size of the standard deviation of the Gaussian random number generator that moves vertices from their location. As seen in Figure 14, even for 20% noise, the algorithm still was capable of finding all subgraphs.

Isogroup Removal We evaluated the speedup and precision of the isogroup removal strategy described in Section 5.1. A test was conducted with results shown in Table 1 where parts of Mexico City,

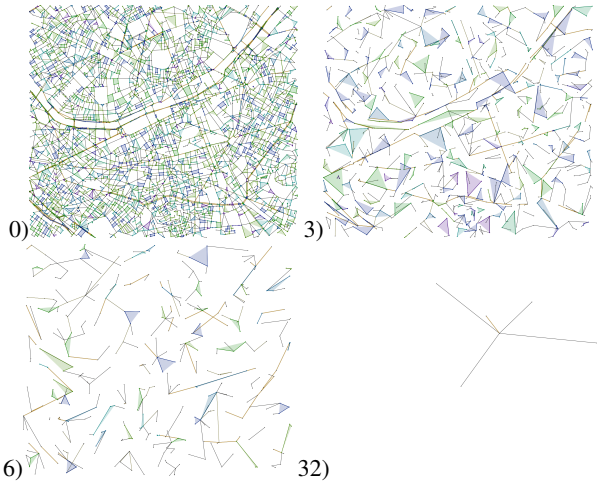

Figure 13: *Successive encoding of a part of Paris into a single terminal node (axiom). The original graph (top left) with color-coded isogroups is compressed and displayed after* 3, 6 *and* 32 *iterations.*

Barcelona, and Indianapolis were used that contain both highly regular and irregular patterns. We performed the complete analysis where all vertices were expanded in each iteration. This was used as the ground truth for these particular cases. We then performed learning in which the number of expanded isogroups was reduced in each iteration.

City	Time [s]		Spdup	Isogroup #		%
	Full	Cut		Full	Cut	
Indy	2	0.3	8	775	54	7
Barcelona	131	1.5	89	5439	338	6
Mexico	2.9k	5.0	584	25k	242	1

Table 1: *Comparison of the geometric graph learning of the* full *expansion of all isogroups and with the* cut *of the least important isogroups.*

The number of the detected isogroups and the speedup are shown in Table 1. The speedup is significant for large datasets. For instance, the total time of analysis for a selected part of Mexico City was reduced from 48 minutes to 5 seconds. The total number of detected isogroups is around 1% of the possible cases. As Figure 15 shows, the reduced learning finds the most frequently repeated isogroups.

The graph in Figure 16 shows the comparison of the frequencies of the detected isogroups in the city of Barcelona for graphs of Order 6. The frequencies of the subgraphs with highest repetitions are similar for the full and the simplified search and the actual differ-

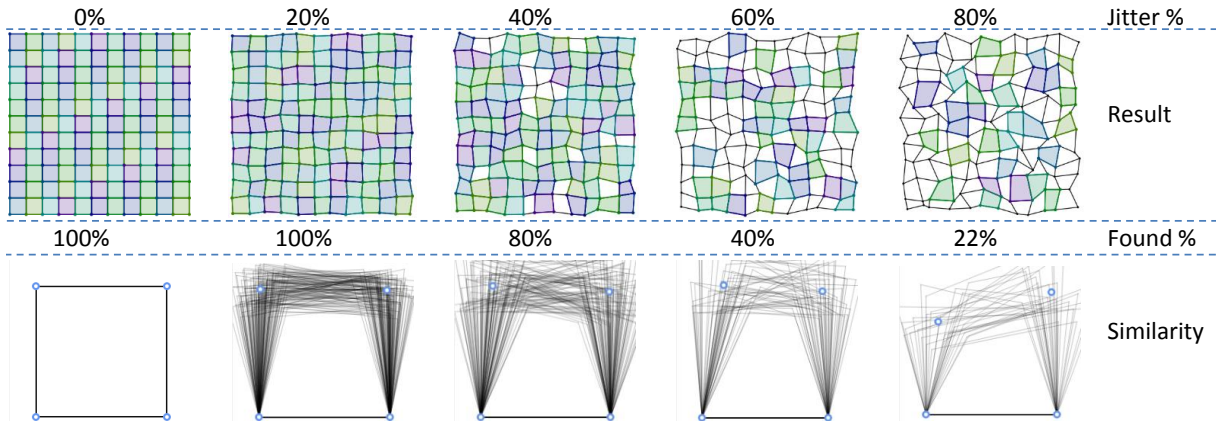

Figure 14: *Noise sensitivity analysis. A regular grid was analyzed and all squares were detected. The same analysis was performed after the locations of graph vertices were randomized by increasing amounts of noise.*

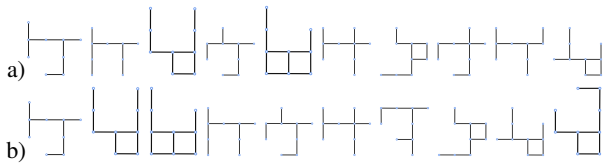

Figure 15: *The full analysis a) provides slightly different results than the analysis in which non-important isogroups are not expanded, b) the difference is in the subgraph ordering.*

ence between both graphs is less than 1%. Although a downside to the speedup is that the optimized pruning detects fewer isogroups (314 vs. 711 in this case), the missed isogroups have less than 10 repetitions and thus do not contribute significantly during encoding.

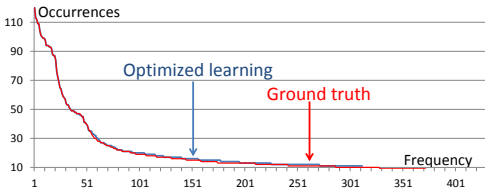

Figure 16: *Comparison of the frequency of occurrences of isogroups detected with the ground truth learning and the optimized version.*

Terminal Replacement Figure 1 shows an example of the grammar terminal node replacement. The highly regular structure of Barcelona was analyzed and frequently repeating regular patterns were automatically detected. These structures were encoded as a terminal symbol of the grammar. Then a round structure (i.e., a plaza) was manually selected and used to replace the most frequently repeated structures.

Complex Example An example in Figure 18 shows stitching of two different cities. In Figure 18 a) both cities were converted to grammars with different user criteria: frequently repeated graphs with near-to-one aspect ratio were learned from Manhattan as well as graphs with a large number of leaves ("spiders"), and highly irregular graphs were detected in Paris. The large repetition reflects the overall style of each city. The rules were manually placed in the

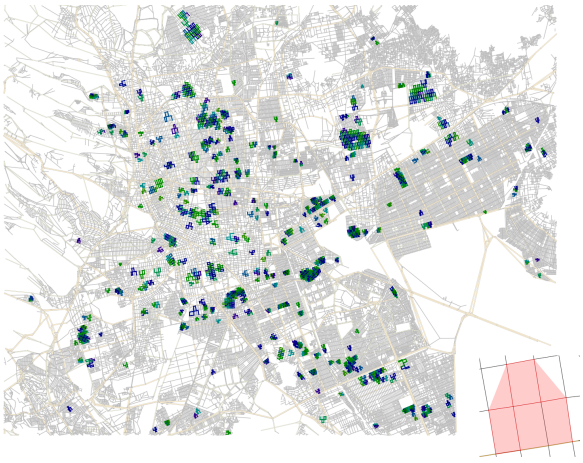

Figure 17: *User-guided analysis. The user selected the pattern in the lower right-hand corner and the system automatically found all its occurrences.*

empty area in Figure 18 b) and the procedural growth automatically connected both examples in Figure 18 c). The overall editing time was under two minutes and most of the time was spent on selecting the suitable part for replacement. The grammar was generated in under one second.

User-guided Analysis starts vertex expansion from a user-defined isogroup as opposed to automatic learning that starts from a simple edge. The user-guided analysis (see the accompanying video) allows user intuition to help analyze the graph; the user can see overall patterns that may be visually important and difficult to capture by automated analysis. It also significantly speeds up the search. When the full analysis is performed, the most time demanding part is the expanding of small graphs with 2-5 vertices. However, if the user selects a larger graph, its expansions are much faster because of the low number of repetitions.

Figure 17 shows large patterns found in a significant portion of Mexico City. This is the largest graph used in the paper (110k edges) and the time for the user-guided analysis was approximately one minute for the pattern in the inset of Figure 17.

Performance evaluation of the GGG learning is provided in Table 2 which shows the number of edges and vertices for each graph,

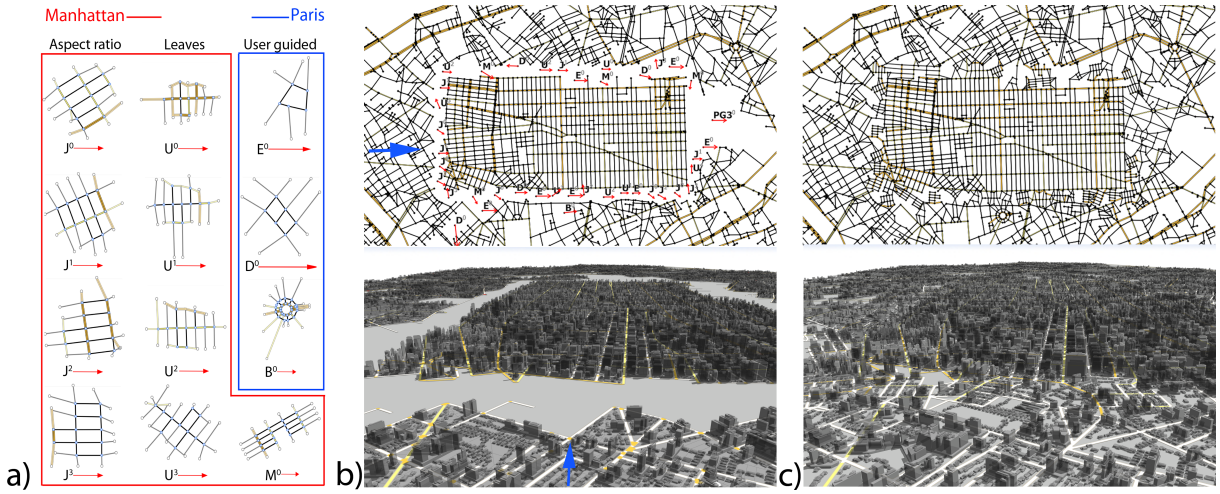

Figure 18: *The cities of Manhattan and Paris were analyzed, encoded, and automatically stitched together. a) Frequently repeated and user-selected subgraphs were found and encoded. Rules from Manhattan (red), from Paris (blue), and the red arrows show the scaling of each instance. The left column shows subgraphs with preferred aspect ratio, the center column shows subgraphs with a larger number of leaves, and the right column shows user guided selection. b) Part of Paris was replaced by Manhattan by adding the rules orientation and location. c) The applied rules stitched both cities together. The bottom row of 3D renderings represent the scene from the camera view indicated by the blue arrow. The street hierarchy and continuity is preserved as well as the overall style.*

the encoding time, the number of generated rules, and the original and encoded size. The size of the input graph is given by the number of vertices and edges. The size of the encoded graph is given by the number of rules, the size and frequency of repetitions of the detected isogroups, and by the size of the axiom.

Fig	$\|V\|$	$\|E\|$	time $[s]$	Rules	Orig. [MB]	Encod. [MB]
1	4.8k	7.1k	22	27	0.10	0.07
11 b)	6.7k	1.0k	20	2696	0.14	0.17
11 c)	6.7k	1.0k	20	2696	0.14	0.17
17	72k	110k	67	1411	1.52	1.47
13	6.7k	10.2k	704	3098	0.17	1.26

Table 2: *Performance and comparison of the examples in the paper. Number of vertices of the input graph $|V|$, number of edges $|E|$, encoding time in seconds, number of generated rules, original size and encoded size in [MB]. The encoded size includes the rules and the axiom.*

8 Conclusions and Future Work

We have introduced geometric graph grammars and provided algorithms to construct them by using vertex expansion of groups of isomorphic graphs. Since the actual number of possible expansions is exponential, we use heuristics that improve the speed of analysis without missing the most frequently repeated subgraphs. After the subgraphs are detected, they are coded into a geometric graph grammar.

Our system is not without some limitations. First it limits graph grammars to those expanding a single node into a new graph. It would be interesting to explore replacing an arbitrary graph with a different graph. The concept of geometric embedding should allow for such operations; however, the task of learning similar graphs would be more challenging. A second limitation is the quantization of the vertex expansion that makes the graph analysis fast, but introduces error in the coding of the vertices that then amplify throughout successive encoding. A third limitation is that new node placement and its connection to the rest of the graph change the appearance of the new graph. Lastly, our system uses edge embedding, where each edge is represented as a line segment. This causes problems when distant parts are connected. Using different edges could improve the visual quality of the result.

There are many possible avenues for future work. We could speed up the isogroup search by implementing different algorithm such as [Cheng et al. 2010]. One of the most significant extensions would be the parametrization of the production rules. Another extension would be the introduction of context sensitivity and environmental responses similar to Open L-systems. In this paper, we have only showed one application area: encoding a road network. An obvious extension is to use geometric graph grammars in 3D. GGGs could be used to explore 3D meshes resulting from other modeling approaches. As triangular meshes are common data structures in computer graphics, a geometric graph grammar analysis could allow mesh encoding and semantic analysis. Graphs are a common dual representation for various semantic analyses in computer vision and the presented approach could be applied to scene analysis.

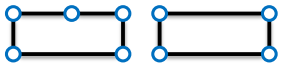

Figure 19: *Two visually similar graphs are detected as different.*

Another avenue for future work comes from the limitation of our approach. Our framework considers the exact topology of the graph so the two graphs from the Figure 19 are considered different even though their shape is exactly the same. If the vertex in the middle is not part of the embedding (i.e., it does not have connection to the rest of the graph), then both graphs could be considered ”visually isomorphic” and this, in effect, would improve analysis.

References

ALIAGA, D. G., ROSEN, P. A., AND BEKINS, D. R. 2007. Style grammars for interactive visualization of architecture. *IEEE Tran. on Vis. and Comp. Graph. 13*, 4, 786–97.

BLOCKEEL, H., AND BRIJDER, R. 2009. Learning non-confluent nlc graph grammar rules. In *Math. Theory and Computational Practice*.

BLOCKEEL, H., AND NIJSSEN, S. 2008. Induction of node label controlled graph grammar rules. In *Intl. Wrkshp. on Mining and Learning with Graphs,*, 1–4.

BOKELOH, M., WAND, M., AND SEIDEL, H.-P. 2010. A connection between partial symmetry and inverse procedural modeling. *ACM Trans. Graph. 29* (July), 104:1–104:10.

CHENG, M.-M., ZHANG, F.-L., MITRA, N. J., HUANG, X., AND HU, S.-M. 2010. Repfinder: Finding approximately repeated scene elements for image editing. *ACM Trans. Graph. 29*, 4, 831–838.

COOK, D. J., AND HOLDER, L. B. 1994. Substructure discovery using minimum description length and background knowledge. *J. Artif. Int. Res. 1*, 231–255.

COOK, D. J., AND HOLDER, L. B. 2006. *Mining Graph Data*. John Wiley & Sons.

DE LA HIGUERA, C. 2010. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, New York, NY, USA.

ENGELFRIET, J., AND ROZENBERG, G. 1991. Graph grammars based on node rewriting: an introduction to nlc graph grammars. In *Graph Grammars and Their Application to Comp. Sci.*, vol. 532. Springer, 12–23.

HAN, F., AND ZHU, S.-C. 2005. Bottom-up/top-down image parsing by attribute graph grammar. In *IEEE International Conference on Comp. Vision*, vol. 2, 1778 –1785 Vol. 2.

JANSSENS, D., AND ROZENBERG, G. 1980. On the structure of node-label-controlled graph languages. *Information Sciences 20*, 3, 191 – 216.

JIANG, C., COENEN, F., AND ZITO, M. 2012. A survey of frequent subgraph mining algorithms. *The Knowledge Engineering Review 28*, 1 (3), 75–105.

KURAMOCHI, M., AND KARYPIS, G. 2007. Discovering frequent geometric subgraphs. *Information Systems 32*, 8, 1101 – 1120.

LIN, L., WU, T., PORWAY, J., AND XU, Z. 2009. A stochastic graph grammar for compositional object representation and recognition. *Pattern Recognition 42*, 7, 1297 – 1307.

NEVILL-MANNING, C. G., WITTEN, I. H., AND MAULSBY, D. 1994. Compression by induction of hierarchical grammars. In *Data Compression Conf.*, 244–253.

PRUSINKIEWICZ, P. 1986. Graphical applications of l-systems. In *Proc. on Graphics Interface '86/Vision Interface '86*, 247–253.

SCHMITTWILKEN, J., DÖRSCHLAG, D., AND PLÜMER, L. 2009. Attribute grammar for 3d city models. In *Proceedings of Urban Data Management Symposium.*

SIROMONEY, G., AND SIROMONEY, R. 1987. Rosenfeld's cycle grammars and kolam. In *Itnl. Workshop on Graph-Grammars and Their Application to Comp. Science*, 564–579.

TALTON, J. O., LOU, Y., LESSER, S., DUKE, J., MĚCH, R., AND KOLTUN, V. 2011. Metropolis procedural modeling. *ACM Trans. Graph. 30* (April), 11:1–11:14.

TALTON, J., YANG, L., KUMAR, R., LIM, M., GOODMAN, N. D., AND MĚCH, R. 2012. Learning design patterns with bayesian grammar induction. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*, UIST '12, 63–74.

VANEGAS, C., GARCIA-DORADO, I., ALIAGA, D. G., WADDELL, P., AND BENES, B. 2012. Inverse design of urban procedural models. In *Proceedings of ACM SIGGRAPH Asia 2012*, ACM Transactions on Graphics.

ŠŤAVA, O., BENES, B., MĚCH, R., ALIAGA, D. G., AND KRIŠTOF, P. 2010. Inverse proc. modeling by automatic generation of l-systems. *Comp. Graph. Forum 29*, 2, 665–674.

ŠŤAVA, O., PIRK, S., KRATT, J., CHEN, B., MĚCH, R., DEUSSEN, O., AND BENES, B. 2014. Inverse procedural modelling of trees. *Computer Graphics Forum 33*, 6, 118–131.

ZHU, S., AND MUMFORD, D. 2007. *A stochastic grammar of images*. No. 4. Now Publishers Inc.