MVA MSc – Internship report
Higher-order Grammars in Computer Vision

Maxim Berman
Supervisor: Nikos Paragios
Center for Visual Computing @ École Centrale

April – August 2014

Abstract

This work introduces the use of Graph Grammars into Computer Vision. After a review of existing grammatical descriptions of images, an introduction to the different formalisms of Graph Grammars is given. A new graph grammar learning method, based on a substructure search algorithm suited for Computer Vision, is presented. The substructure search algorithm and grammar learning method performance are tested on images of building facades in the ECP facades databases.

Acknowledgements

I heartily thank Professor Nikos Paragios for his support and supervision during this research internship, for the trust I was given during this work and for the great resources put at my disposal at the CVC laboratory at École Centrale Paris. I also wish to thank the people in charge of the MVA Masters at ENS Cachan for the organization of the internships, both on the pedagogic and on the administrative side, and Natalia Leclercq, the very efficient and supportive secretary of the CVC lab.

Contents

Introduction

Essential to intelligent technologies omnipresent in our daily lives, from personal digital assistants and monitoring systems to life-saving biomedical devices, Computer Vision has become a key challenge of the 21th century. Thanks to the massive research effort in the field and the exponential increase in available processing power, the analysis of images and videos has seen great progress in last decades, and state-of-the-art object detection approaches human capacity.

However, competitive detection systems often rely only on low-level image representations. Developing an efficient use and learning of high-level image semantics in order to assist the Computer Vision task, towards whole-scene understanding, remains a subject of active research.

This search for a concise representation of the composition of images explains the recent emergence of Grammars in the field. Introduced in the 50s by the linguist Noam Chomsky as a model of human languages [1], formal grammars, that describe languages by a finite set of operations on symbols, have quickly been adopted by computer scientists to formalize computer languages, and have found applications in many other fields: biology [11], botany [12], architecture [19], computer graphics…

Some categories of images can be naturally associated to a description in terms of a grammar. Each particular instance of an object contained in the category can be associated to a derivation tree of the grammar; therefore, the grammar serves both as a constraint and an aid to the computer vision task, such as the recognition of the objects contained in the picture.

The different approaches of introducing formal grammar models into Computer Vision lead to simple and often incomplete models of image semantics: while suited to model objects, grammars fail at describing whole scenes. This can be thought of as a limitation of grammars themselves, because grammar derivations generate derivation trees and can therefore only lead to tree-like constraints in an image, while some other Computer Vision models, such as Deformable Part Models, introduce graph-like, cyclic constraints in images.

This observation calls for the search of an extension of grammar models suited for describing not only tree constraints, but entire graphs. *Graph Grammars*, introduced in the late 60s, are natural candidates. These grammars can be seen as an extension of formal grammars, and describe families of graphs using a finite set of graph transformation rules. The symbols manipulated by Graph Grammars are graphs themselves. This leads to a great expressive power of graph grammars.

This work reviews the main uses of grammars in Computer Vision and introduces Graph Grammars as a new language of model description. First, in Section 1, the formalism of formal grammars is presented, along with former uses of grammars and grammar learning methods in the field. In Section 2, the formalism of Graph Grammars is presented. Finally, Section 3 presents the new methods and results obtained during this internship.

1 Grammars: formalism and uses in Computer Vision

1.1 Formal Grammars

A formal grammar describes transformations of a family of symbols according to a fixed set of rules. It is composed of a set of symbols $\mathcal{A}$, a set of nonterminal symbols $\mathcal{N} \subset \mathcal{A}$, a *starting symbol* $\mathbf{S} \in \mathcal{N}$, and a set of *productions* $\mathbf{P}$ of the form

$$\mathbf{X} \Longrightarrow \mathbf{Y_1 Y_2} \dots \mathbf{Y_n} \tag{1}$$

describing a substitution of a nonterminal symbol $\mathbf{X} \in \mathcal{N}$ into a succession of symbols $\mathbf{Y_1}, \dots, \mathbf{Y_n} \in \mathcal{A}$. A grammar derivation starts with the starting symbol $\mathbf{S}$ and applies the substitutions described by the grammar productions iteratively until the obtained word contains only terminal symbols.

This is best understood through simple examples. Consider a grammar $\mathbf{G}_1$, operating on the alphabet $\{a, b, \mathbf{S}\}$, its starting symbol $\mathbf{S}$ being its only nonterminal, comprising two productions

$$\mathbf{S} \to a\mathbf{S}b \tag{p1}$$
$$\mathbf{S} \to ab. \tag{p2}$$

One derivation of grammar $\mathbf{G}_1$ is

$$\mathbf{S} \overset{\text{(p1)}}{\Longrightarrow} a\mathbf{S}b \overset{\text{(p1)}}{\Longrightarrow} aa\mathbf{S}bb \overset{\text{(p2)}}{\Longrightarrow} aaabbb. \tag{2}$$

The language generated by grammar $\mathcal{L}(\mathbf{G})$ is the set of words obtained from every possible derivation of the grammar starting with starting symbol S. In our example, the language generated by $\mathbf{G}_1$ can easily be inferred to be the set of words starting with a symbols followed by b symbols repeated in equal number, that is

$$\mathcal{L}(\mathbf{G}_1) = \{a^n b^n \mid n \geq 1\}. \tag{3}$$

Let us consider a second example: $\mathbf{G}_2$ operates on the alphabet $\{a, b, c, \mathbf{S}, \mathbf{B}\}$ and has four productions

$$\mathbf{S} \to a\mathbf{B}\mathbf{S}c \tag{p1}$$
$$\mathbf{S} \to ab. \tag{p2}$$
$$\mathbf{B}a \to a\mathbf{B}. \tag{p3}$$
$$\mathbf{B}b \to bb. \tag{p4}$$

This time the grammar has 3 terminal symbols a, b, c and 2 nonterminal symbols—in upper case by convention: the starting symbol $\mathbf{S}$ and symbol $\mathbf{B}$. One derivation of this grammar is

$$\begin{aligned}\mathbf{S} &\overset{\text{(p1)}}{\Longrightarrow} a\mathbf{B}\mathbf{S}c \overset{\text{(p1)}}{\Longrightarrow} a\mathbf{B}a\mathbf{B}\mathbf{S}cc \overset{\text{(p2)}}{\Longrightarrow} a\mathbf{B}a\mathbf{B}abccc \overset{\text{(p3)}}{\Longrightarrow} a\mathbf{B}aa\mathbf{B}bccc \\ &\overset{\text{(p3)}}{\Longrightarrow} aa\mathbf{B}a\mathbf{B}bccc \overset{\text{(p3)}}{\Longrightarrow} aaa\mathbf{B}\mathbf{B}bccc \overset{\text{(p4)}}{\Longrightarrow} aaa\mathbf{B}bbccc \overset{\text{(p4)}}{\Longrightarrow} aaabbbccc.\end{aligned} \tag{4}$$

It is straightforward to infer that the language generated by this second grammar is

$$\mathcal{L}(\mathbf{G}_2) = \{a^n b^n c^n \mid n \geq 1\}. \tag{5}$$

The examples of grammars $\mathbf{G}_1$ and $\mathbf{G}_2$ provide some fruitful insight on the computational properties of grammars. The languages generated by the two grammars look quite similar. However, the presence of more than one symbol at the left side of productions (p3) and (p4) fundamentally differentiates the two grammars: $\mathbf{G}_2$ uses the *context* of symbols in the production rules (the symbols that are near the nonterminal symbols), as opposed to $\mathbf{G}_1$—which is therefore called *context-free*.

This difference has crucial implications. Consider the

Parsing Problem. Given a word w and a grammar $\mathbf{G}$, is $w \in \mathcal{L}(\mathbf{G})$?

While any context-free grammar can be parsed in cubic time using a generic algorithm [17], no general parsing algorithm exists for unrestricted grammars.

Another common problem of formal grammars, central to this work, is the

Inference Problem. Given a set of words $w_1, w_2 \ldots, w_n$, find a grammar $\mathbf{G}$ such that $w_1, \ldots, w_n \in \mathcal{L}(\mathbf{G})$—under certain simplicity assumptions on $\mathbf{G}$.

While inference is rather a class of problems and can be formalized in different ways, it is clear that inferring an unrestricted grammar from a set of examples is close to impossible: in fact, example $\mathbf{G}_2$ shows that the convoluted productions that have to be used in order to infer the apparently simple language $\{a^n b^n c^n\}$ are difficult to guess even for humans. Therefore, grammar inference methods are limited within restricted classes of grammars.

1.2 Use of grammars in Computer Vision

1.2.1 Object detection grammars

Object detection grammars were introduced by Felzenswalb and McAllester [7] in order to give a grammar formalism of compositional models in Computer Vision. It was proposed as an evolution to Deformable Part Models, suited for the recognition of objects with high compositional variability. The aim of object detection grammars is category-level object detection. Objects of interest in input images have to be detected, given a bounding box, and assigned to one of few object categories. Object detection is one of the core problems of Computer Vision today, and is still far from solved.

One of the main challenges of object detection is the great variation in appearance of objects within object classes. An object can endure a deformation by having the relative position or orientations of its parts changes, leading to a smooth change in appearance, or even to a non-smooth change if self-occlusions come into play. Other "soft body" objects, such as cats, may be subject to non-rigid deformations. Viewpoint variations also change the appearance of objects:

when these variations are large, they can change the shape of objects completely, making it impossible to model the variation as a smooth deformation.

Deformable Part Models [5] (DPMs) addresses some of this appearance variability by representing objects as a deformable arrangement of their parts. For instance, a bike can be thought as a deformable arrangement of its wheels, saddle and handlebar, and a person can be similarly decomposed into its body parts. Hence, by learning the appearance of each subparts—this appearance being for instance captured as a Histogram-Of-Gradient filter—and the relative positions of these subparts, one obtains an object model robust to deformations [6]. The use of star-shaped DPMs (each object being positioned relative to one central object) or tree-linked DPMs leads to efficient learning of such models using dynamic programming.

When more dramatic viewport changes come into play, a single deformable model struggles with fitting all the views of an object. For instance, bikes have very different shapes when seen from the front and seen from the side. One solution is to use a mixture of deformable models [6], as shown on Figure 1. Each component of the mixture corresponds to a different view of the object. The learning uses a latent SVM approach in order to select the appropriate component as a latent variable.

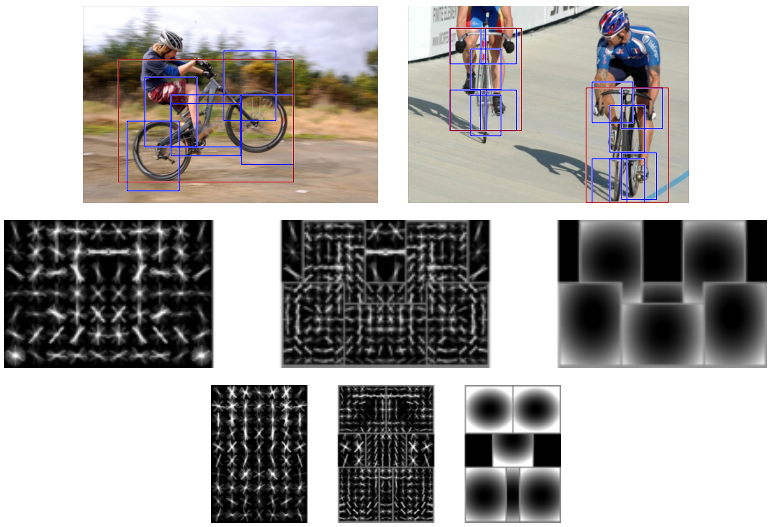


Figure 1: Detection of bicycles using a 2-component mixture of deformation models. The first component captures sideways views and the second front views of bicycles. Illustration taken from Felzenszwalb et al. [6]

Mixture deformation models performs at state of the art level on various object detection challenges; however, it falls short of addressing all sources of appearance variability. Indeed, mixture DPMs cannot address the compositional nature of many rich object classes. A person, for instance, can present infinite variations in the appearance of its parts, simply by wearing a hat, glasses or shoes, or playing an instrument. Two instances of the same object could moreover have a different number of subparts, as is the case for buildings with different numbers

of windows, or trains with different numbers of carriages. The change of subparts does not constitute a smooth deformation: it can therefore not be addressed in a single-component DPM. Given the exponential number of combinations that may be formed by choosing the subparts, it is not possible to devote one component of a mixture DPM to each possible combination of parts either.

Grammar models are an elegant way of addressing the compositional complexity of objects. A grammar model describes an object as a composition of its parts: the placement of a part, like a face, may optionally give rise to the placement of a subpart, like a hat or a tuba mask; the rules describing the possible combinations are contained in the grammar productions. Figure 2 gives an informal illustration of the use of a grammar to describe a person.

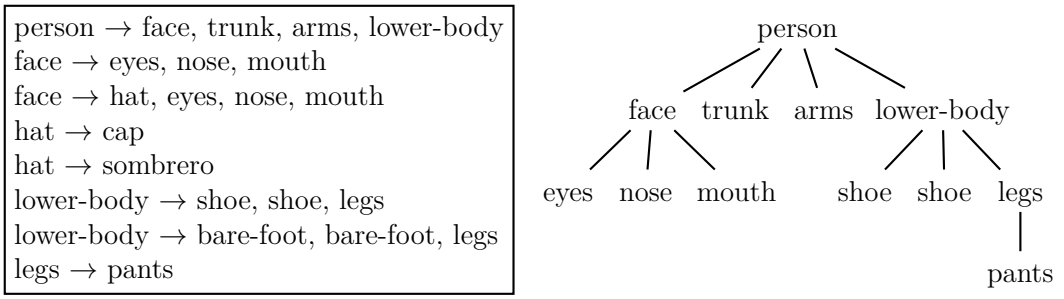


Figure 2: Illustration of a grammar describing a person. Left: informal formulation of productions in a person grammar. Right: a derivation tree of this grammar, corresponding to the particular instance of a person.

The formalization of this idea leads to object detection grammars [7], which are very similar in nature to the grammar formalism described in Section 1.1. The grammar operates on symbols according to its production rules, and has a set of nonterminal symbols $\mathcal{N}$ and a set of terminal symbols $\mathcal{T}$. A grammar derivation expands nonterminals into terminal symbols. Terminal symbols are the elementary parts that are not decomposed into new subparts, such as an eye in a face or a wheel on a bike. They are associated with a learned appearance, represented e.g. by a HOG filter. Nonterminals can be decomposed into different subparts—e.g. a face could be decomposed into eyes, a mouth and a nose, or expanded into different possible appearances—e.g. a mouth nonterminal could be replaced by a smiling mouth or a frowning mouth.

In order to introduce geometry in the models, the notion of *placed symbol* is introduced. A symbol $Y \in \mathcal{N} \cup \mathcal{T}$ may be placed in an image at a location $\omega \in \Omega$, where Ω is a set of possible locations. The productions of the grammar are take the form

$$X(\omega_0) \xrightarrow{s} \{Y_1(\omega_1), \ldots, Y_n(\omega_n)\}, \tag{6}$$

replacing a placed nonterminal X by a set of placed symbols $Y_1, \ldots, Y_n$. The right-side of the productions is not ordered, on the opposite of the formal grammar formalism. Moreover, each production is weighted by a score s. The number of possible positions $\omega \in \Omega$ being large, the number of productions is very large

or infinite. However we can group parametrize families of productions together, therefore speaking of a *production schema* of the form

$$X(\omega_0) \xrightarrow{s(\omega_0,\omega_1,\ldots,\omega_n)} \{Y_1(\omega_1),\ldots,Y_n(\omega_n)\}, \tag{7}$$

where the score of a production depends on the position of the symbol being replaced and the new placed symbols.

The detection of an object using a detection grammar is done by maximizing a score, as in a DPM; however, the optimization also spans all possible derivations of a grammar. Each non-terminal expansion is given the score associated to its production, as seen above; each placement of a terminal is given a score corresponding to the affinity of its appearance filter to its position in the image, as depicted on Figure 3; the score of a derivation is simply the sum of the scores of its expansions and of the placement of its terminals.

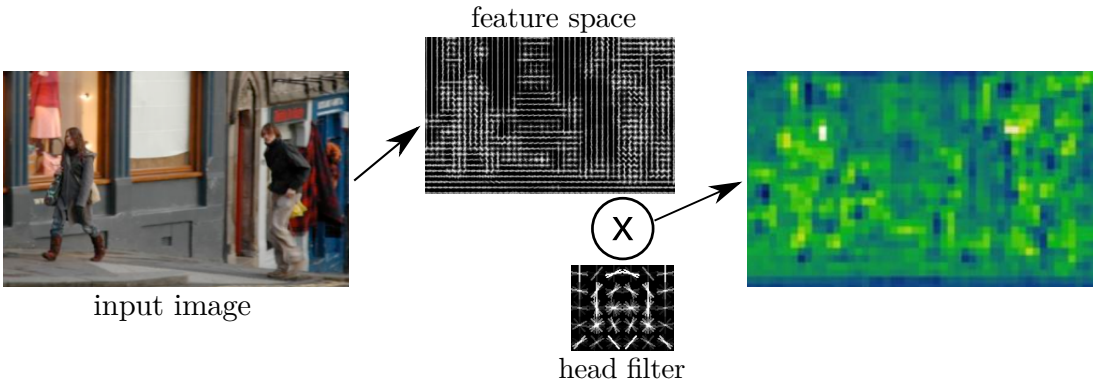


Figure 3: Affinity of a *head* filter on an image. Bright regions in the affinity map (on the right) correspond to likely heads candidates. Illustration taken from [6]

Isolated deformation grammars Isolated deformation grammars are a special class of object detection grammars that allow for efficient learning. In these grammars, subpart placement production schemas

$$X(\omega) \xrightarrow{\beta} \{Y_1(\omega+\delta_1),\ldots,Y_n(\omega+\delta_n)\} \tag{8}$$

are isolated from deformation production schemas

$$X(\omega) \xrightarrow{\beta(\delta)} \{Y(\omega+\delta)\} \tag{9}$$

allowing for the use of dynamic programming techniques and, if the deformation cost $\beta(\delta)$ is quadratic in $\beta(\delta)$, distance transforms, for efficient object detection—refer to [6] for details.

The appearance filter of each subpart is learned independently from the derivation of particular instances. This is a huge improvement over a mixture DPM where each component of the mixture has to be learned independently,

and there is no shared knowledge between subparts in different components, for instance the appearance of a face in different components of a mixture DPM person model.

The grammar model for person detection presented in article [7] shows high performance on the PASCAL benchmark [4]. The grammar parameters, appearance filters and production scores, are learned from examples. However, the grammar is defined by hand: the working system defines a person as having 6 main parts and a possible *occluder* part, as seen on Figure 4 (these parts have subparts themselves). The derivations are made from the head to the bottom: is the lower-body of a person is occluded, then the occluder part is added and terminates the derivation.

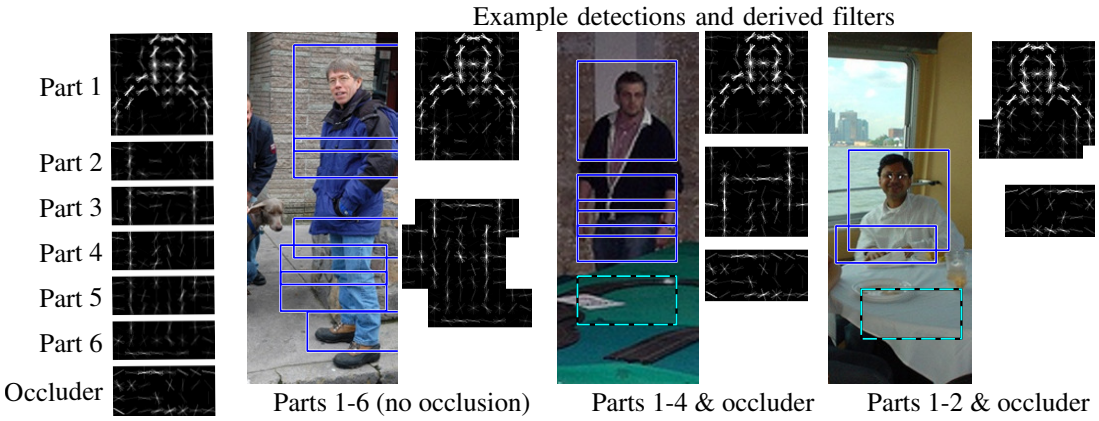
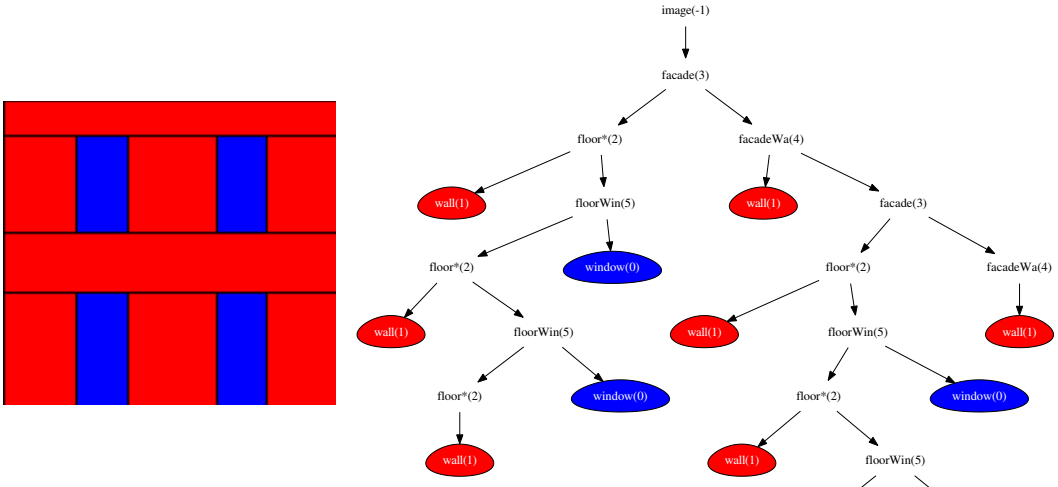


Figure 4: Overview of the first layer of the grammar model defined in [7] (image taken from the article). The occluder part is derived when the lower-part of a person is occluded.

1.2.2 Split grammars

Split grammars have been introduced as a procedural description tool in architecture by Peter Wonka [19], building upon Stiny's early work on Shape grammars [14]. Split grammars have been successfully used in Computer Vision, in the field of architectural buildings parsing. On the opposite of the object detection grammars discussed previously, split grammars do not consider relations between high-level objects such as wheels in a bike or windows on a building. Instead, they start from a low-level shape, such as a 2D rectangle, and describe the successive splits that must be applied to this shape in order to obtain the shape of a particular object.

Work on inference The first works on the use of binary split grammars in facade parsing involved manually defined grammars in the recognition process. A grammar description being restricted to a particular architecture, such as haussmannian facades, this approach is tedious when it comes to accommodating a new architectural style. This motivates the work in automatic inference of split



Parent	Children		Split
Axiom(W,H)	Facade(0,0,W,H)		None
Facade(0,Y,W,H)	Floor(0,Y,W,h)	Fa-Wall(0,Y+h,W,H-h)	Y:h
Fa-Wall(0,Y,W,H)	Wall(0,Y,W,h)	Facade(0,Y+h,W,H-h)	Y:h
Floor(X,Y,W,H)	Wall(X,Y,w,H)	Fl-Win(X+w,Y,W-w,H)	X:w
Fl-Win(X,Y,W,H)	Window(X,Y,w,H)	Floor(X,Y,W-w,H)	X:w

Figure 5: Example of a building facade (left), corresponding derivation tree (right) using a simple Binary Split Grammar whose productions are listed (bottom table). Each production splits a rectangle node in a horizontal or vertical way and defines new subrectangle regions on each side of the split. Illustration taken from Teboul et al. [16].

grammars from a set of example buildings sharing the same architecture. Two main approaches have considered in the field of binary split grammars learning.

- Weissenberg et al. [18] developed tools to generate procedural grammars directly from image of buildings, by detecting changes of regions in the images and inferring a split grammar from this low-level knowledge. The generated grammar has low compression rate (high redundancy of inferred production rules) but the method is fully automatic with no prior assumption on the structure of buildings

- R. Gadde (CVC, École Centrale) is developing a split grammar inference method which starts with a basic generic split grammars describing buildings, and particularizes this grammar for a given architecture by finding redundant subtrees in building derivations and replacing them with new grammar rules. This produces a cleaner grammar than the previous method, but does use prior knowledge on the structure of buildings.

2 Graph grammars

In the previous section, we have seen that grammars have been used as an image specification tool in different areas of Computer Vision. Both in object detection grammars and in binary split grammars, grammar productions engender a derivation tree suited to a particular instance of an object. This derivation tree is used as a guide and a constraint to the recognition task. One shortcoming of this approach is that a derivation will necessarily engender a tree; while this representation seems very suited to some classes of compositional objects, it seems impossible to create a tree-representation of an entire natural scene. Moreover, the models described before present no *structural innovation*. A repetition of carriages on a train will not be detected as such in order to infer some general rule that a train is composed of an undetermined number of repeated carriages.

Graph grammars are more sophisticated than formal grammars. Instead of producing a derivation tree from symbol transformations, graph grammars describe transformations that affect a graph: graph grammars are *graph rewriting systems*. Generally speaking, graph grammars search for all occurrences of a subgraph $\mathcal{H}$ in a graph $\mathcal{G}$ and replace them by a new subgraph $\mathcal{H}$, according to one of the grammar productions. The productions do not simply describe the replacement of subgraphs by new subgraphs: they also include *connection instructions* that describe how the new subgraph $\mathcal{H}'$ should be connected to the neighbours of the old subgraph $\mathcal{H}$.

There are two complementary approaches to graph grammars: node rewriting systems, in which a node in a graph is replaced by a new subgraph; and (hyper)edge rewriting systems, which replaces edges in a graph (or hyperedges in a hypergraph) by a new subgraph. Either systems are described in the two following subsections.

2.1 Definition

2.2 Node Replacement Grammars

Node Replacement (NR) grammars are a class of graph grammars that act by replacing a node n in a graph $\mathcal{G}$ by a new subgraph $\mathcal{H}$. There are different classes of node replacement grammars, having different language expressiveness: these classes differ by their *connection instructions*, which specifies how the new subgraph $\mathcal{H}$ should be connected to the former neighbours of n in $\mathcal{G}$.

Node-label Controlled (NLC) replacement grammars are a simple class of NR grammars. The connections instructions take the form (X, Y), where X is a label of a node in $\mathcal{G}$, and Y is a label of a node in the new subgraph $\mathcal{H}$. If the node n in $\mathcal{G}$ is being replaced by $\mathcal{H}$, then the former neighbours of n having label X should be linked to the nodes of $\mathcal{H}$ having label Y.

The example of Figure 6 illustrates the behavior of NLC grammars. One starts with the start symbol X. Applying production ② stops the derivation (there are no nonterminals left). Applying production ① and then ②, one the top graph of Figure 7. Applying ① two times and then ①, one obtains the

bottom graph of Figure 7. More generally, one can infer that this grammar generates all chains $(abc)^n$ for $n \geq 1$ with extra edges between all b-labelled nodes.

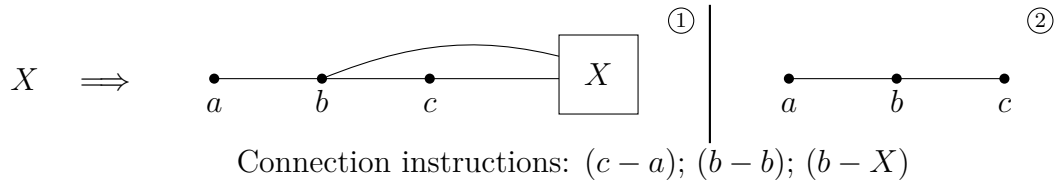


Figure 6: A node-label controlled graph grammar. The vertical bar is a shorthand *or* notation: the nonterminal X can be replaced either by the subgraph on the left or by the subgraph on the right; hence the grammar has two production rules.

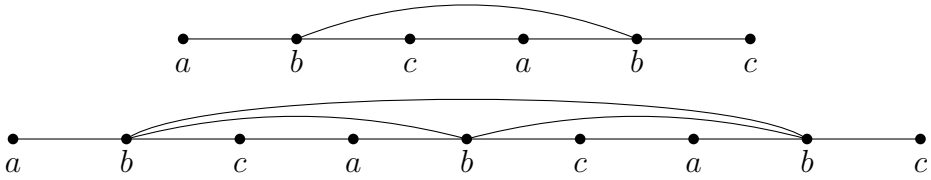


Figure 7: Two graphs generated by the NLC grammar described in Figure 6.

NLC grammars operate on undirected, node-labelled graphs. More general classes of NR grammars operate on graph with directed and labelled edges. Moreover, when replacing node n in $\mathcal{G}$ by subgraph $\mathcal{H}$, the connection instructions become more powerful if they can refer to specific nodes in $\mathcal{H}$ instead of only refering to the labels of the nodes in $\mathcal{H}$: this way, it becomes possible to distinguish between two nodes in $\mathcal{H}$ sharing the same label. The most general class of NR grammars that incorporate all these extensions is the class of edNCE grammars, standing for edge-labelled directed Neighborhood Controlled Embedding. Without going into further details, Figure 8 illustrates in a graphic notation the production of such a grammar.

2.3 Hyperedge Replacement Grammars

A complementary approach to node replacement grammars are hyperedge replacement grammars. Generally, they operate on hypergraphs, a generalization of graphs where edges are no longer binary relations between two nodes, but k-ary relations between k nodes. An edge consists of an ordered list of nodes $(n_1, n_2, \ldots, n_k)$ and a label A. If a k-ary edge $(n_1, n_2, \ldots, n_k)$ is at the left side of a production, the graph on the right side of the production will comprise *handles* $1, \ldots, k$ indicating the connections to nodes $(n_1, n_2, \ldots, n_k)$.

Figure 9(a) shows an example of a hyperedge replacement grammar. The edges are represented by squares, and the connection of hyperedges to their nodes

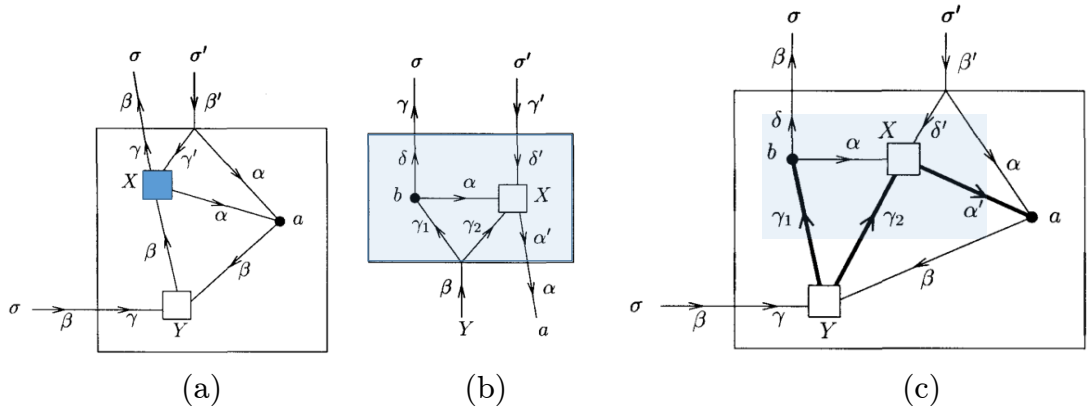
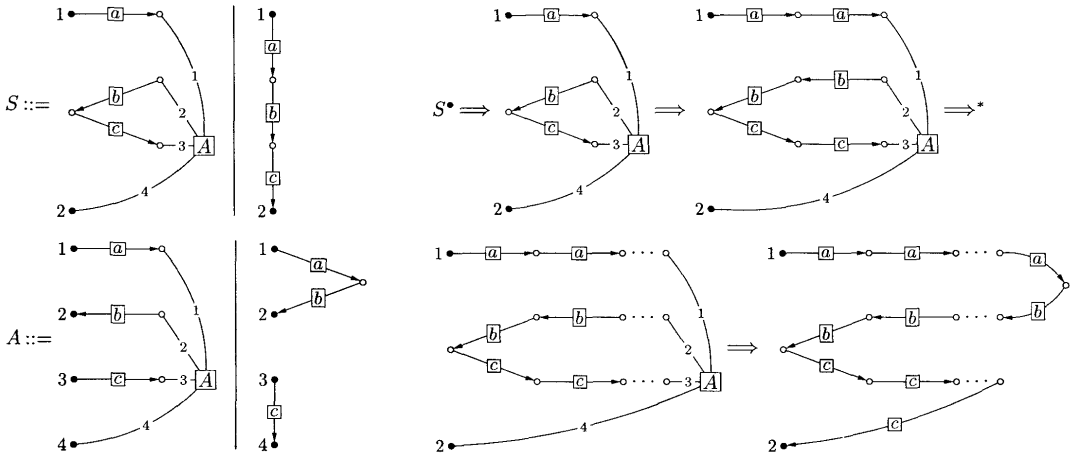


Figure 8: An example of a production of an edNCE grammar: the graphs (inside rectangular boxes) are augmented with their connection instructions (outside rectangular boxes). X is a nonterminal node. In the replacement of node X in graph (a) by subgraph (b), the whole box around graph (b) replaces the node X produces graph (c). The connections to the box are replaced by their continuation inside the box. One obtains graph (c) after this production. Refer to [13] for further details.

is numbered. Figure 9(b) shows a derivation of the grammar. The generated graph languages consists of chains $a^n b^n c^n$, $n \geq 1$.



(a) The grammar's 4 productions.

(b) A derivation in the grammar.

Figure 9: A hyperedge replacement grammar generating chains $a^n b^n c^n$, $n \geq 1$; S and A are nonterminals, S is the start symbol (extracted from [13]).

3 Methods and results

3.1 Research direction

Object detection grammars, introduced by Felzenszwalb et al. and described in Section 1.2.1, constitute an major step toward semantic-assisted object detection. A model of the semantic structure of a class of images, described with a few grammar rules, is able to support a Computer Vision task. The approach does however have its limits:

1. It requires a manually-defined grammar;

2. The grammar generates derivation trees that describe the constraints between the subparts of the object. It seems unlikely that these tree-constrains can be appropriate to describe whole scenes; the method is suited for some classes of objects but not for whole-scene understanding, since the constraints can only follow a tree;

3. The grammar uses no recursive rules: for instance, there can be no rule describing a building as a succession of n windows, for an undetermined n.

The presence of an *occluder* element in the person detection grammar, as seen in Figure 4, is a good illustration of point 2. The occluder is necessary to handle occlusions of the bottom part of a person behind an object; yet it has no semantic meaning and is not part of a person; it is introduced artificially for a better efficiency of the grammar model.

Split grammars, as presented in Section 1.2.2, use grammars in a different way, as a shape description language. While they turn out to be appropriate for some object classes, such as building facades, they are based on splitting of rectangular shapes and seem therefore difficult to adapt in the case of arbitrary-shaped natural objects. Moreover, as they generate tree-structured graphs, they present the same shortcomings as object detection grammars, and the constraints of the models follow the structure of the tree. For instance, in the case of a building facade, the vertical alignment of windows has to be introduced as a manual requirement in the model and can not be enforced by the grammar only.

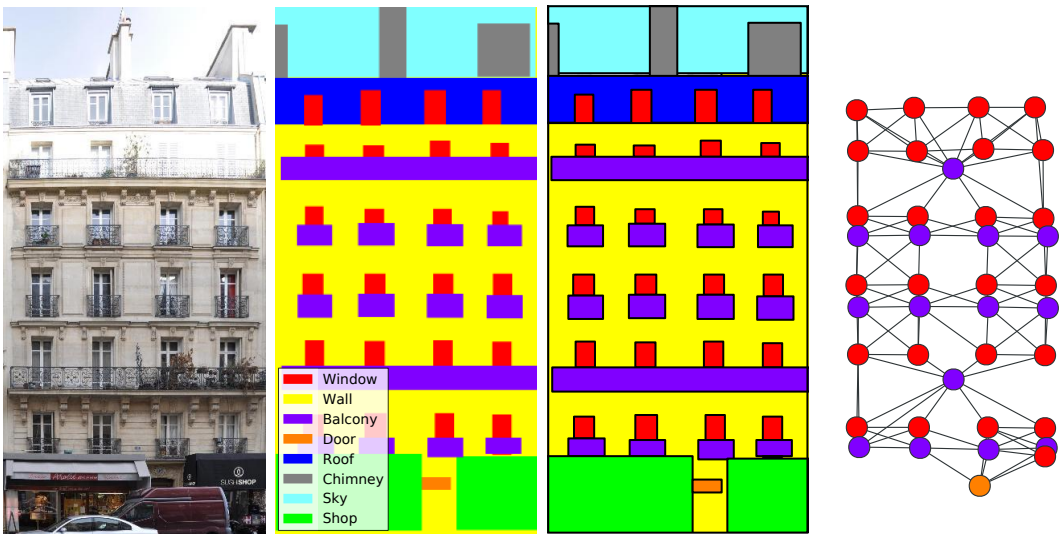
We have seen how graph grammars are more powerful than formal grammars as a graph rewriting systems. While formal grammars can describe trees through their parse tree, graph grammars operate directly on graphs. Graph grammars are therefore natural candidates for describing graphs appearing in Computer Vision, such as Deformable Part Models.

We have seen how grammar inference is crucial in automatizing the use of grammars in Computer Vision: it is not possible to rely on manually-defined grammars on new problems. This is even more true in the case of graph grammars where it is difficult for a human to design a graph grammar suited for a particular problem. While graph grammars have been quite extensively studied from the point of view of their expressiveness as a language, the automatic learning of graph grammars that would suit a particular family of graphs is to a large extent

an open problem. For this reason, the internship has been focused on designing a method of automatic generation of a graph grammar from a family of graphs.

3.2 Working graphs

This work on graph grammars have been done using images of building facades. Indeed, facade buildings are relatively simple objects with repeating aligned structures and appear as a good starting point for the introducing graph grammars in Computer Vision. Moreover, the labelling of many facades is already available through the ECP Facades Database [15].



(a) image (b) labelling (c) conn. components (d) graph

Figure 10: From a facade image to a graph.

Graphs are built directly from the labelled images: by means of a connected components analysis, the rectangular objects are localized in the pictures. Each *window*, *balcony* and *door* object is then associated with a node in the graph. The node is labelled with the nature of the object it represents. Then edges between close objects are added to the graph. Different methods were considered:

1. Connecting all objects which centers are closer than a threshold,
2. Connecting an object to its k nearest-neighbours,
   - using its distance to closest centers
   - using the distance between its shape rectangle and the shape rectangle of other objects.

Method 1 produces densely connected subgraphs and high degree nodes in regions where there are more objects. Since the running time of the algorithms developed in this work degrades poorly with the graphs degrees, a nearest-neighbour method, which allows to control the maximum degrees of the nodes,

was adopted instead. The use of the rectangle-distance, that measures the closest distance between the two shape rectangles, instead of the distance between centers, allow us to capture more information on the adjacencies between elements: see for example how the running balconies connect to all windows on the floor in the graph of Figure 10. The nodes are colored according to their nature, and the same colour scheme will be used in the rest of this document.

3.3 References on graph grammar learning

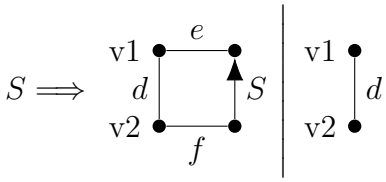
There is little literature on the subject of graph grammar learning;

- Fürst et al. [8] use a parser-controlled process to infer a grammar from a set of positive and negative example graphs. The algorithm starts with an elementary grammar generating all positive graphs, and generalizes its productions of this grammar as long as none of the negative examples can be generated by the learned grammar,

- Kukluk et al. [9, 10] detect repeating substructures in one example graph, and infer a production from these repeating substructures. The process is iterated until the productions no longer *compress* the information of the graph.

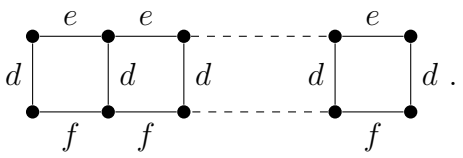
The second approach is simpler: it uses no graph grammar parser, and there is no need to define any generalization operation on graph grammar productions, as in the first approach. For these reasons, it was the starting point of this research.

The aim of this graph grammar learning method introduced by Kukluk et al. [10] is to infer an edge replacement grammar from an example graph. One example of an edge replacement grammar possibly inferred by the method consists of the productions



(10)

where $v1$ and $v2$ mark the two handles of the replacement symbols. This grammar leads to the graph language containing the single d-labelled edge and the chains



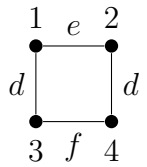
(11)

The algorithm proceeds in three steps:

1. Identification of repeating substructures:

   Using a substructure discovery algorithm based on the work of Cook and Holder [2], a repeating subgraph $\mathcal{S}$ is identified in the example graph $\mathcal{G}$.

On our previous edge grammar example, this repeating subgraph could be



. (12)

Two instances of this substructure in the graph should have no more than two vertices in common.

2. Inference of a new grammar rule:

   The edges where the different instances of the substructure connect are identified. On our previous example, instances of (12) are connected by their $(2, 4)$ edges to the $(1, 3)$ edge of other instances. This identified connection is inferred as a new grammar rule.

3. Compression of the graph:

   Every instances of the best substructure found are replaced by single vertices: this operation can be reversed using the new grammar rule. The obtained graph is the graph $\mathcal{G}$ compressed by the substructure $\mathcal{S}$, noted $\mathcal{G}|\mathcal{S}$.

Steps 1–3 are repeated and new grammar rules are found at each iteration; the algorithm continues until there is no more information contained in graph $\mathcal{G}$ than in graph $\mathcal{G}|\mathcal{S}$, according to some information measure taking the description of the grammar rules into account.

3.4 Substructure search

3.4.1 Algorithm

A key of the graph grammar learning procedure described earlier is the search of repeating patterns in the graph. This has been one main concern of this work. The approach developed here draws inspiration from the work of Cook and Holder [2].

A *repeating substructure* $\mathcal{S}$ is a connected graph isomorphic to multiple subgraphs $\mathcal{S}_1, \ldots, \mathcal{S}_n$ of a host graph $\mathcal{G}$. $\mathcal{S}_1, \ldots, \mathcal{S}_n$ are the *instances* of $\mathcal{S}$ in $\mathcal{G}$.

The smallest substructures of a graph are single nodes. There are as many 1-node substructures as the number of different labels in the graph. These are the initial *parent substructures* of the algorithm.

Bigger substructures are found by *growing* parent substructures, one supplementary vertex at a time. The subgraphs obtained by extending each instance of the parent substructures along an outgoing edge in the graph $\mathcal{G}$ are grouped into new isomorphic substructures with one supplementary vertex. The score of these bigger substructures is computed according to some measure. Only the top-scoring substructures are kept for the following iteration of the algorithm (beam search). Algorithm 1 summarizes this procedure.

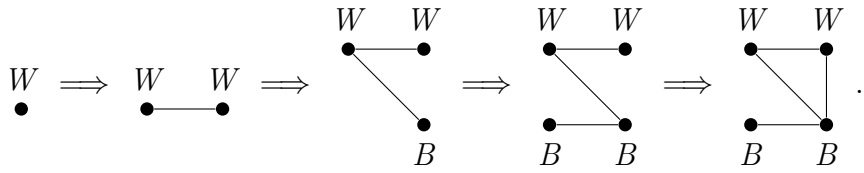
Algorithm 1 Substructure Discovery

```
1: procedure findSubstructures(Graph 𝒢, int maxDepth, int beamWidth, int maxSharedVtx)
2:     ▷ Initialize substructures to single nodes
3:     L ←new substructure list
4:     for all distinct vertex label l in 𝒢 :
5:         I ←single vertices labeled l
6:         insert substructure (l) with instances I into L
7:     bestSub ←first substructure in L                ▷ best substructure initialization
8:     ▷ Grow substructures
9:     for depth ≤ maxDepth :
10:        L_new ←new substructure list
11:        for all 𝒮 in L :
12:            I ←instances of 𝒮
13:            I_new ←All subgraphs obtained by extending instances in I along neighbouring edges in 𝒮
14:            for all inst in I_new :
15:                if inst is isomorphic to a substructure 𝒮 in L_new :
16:                    if no instance of 𝒮 share more than maxSharedVtx with inst :
17:                        insert inst into the instances of 𝒮
18:                else
19:                    insert new substructure isomorphic to inst into L_new
20:                    insert inst into the new substructure's instances
21:        scores ← evaluateSubstructures(g, L_new)
22:        ▷ Pruning
23:        prune substructures with only 1 instance in L_new
24:        order L_new by scores
25:        keep only beamWidth first substructures in L_new
26:        replace bestSub if a higher-scoring substructure has been found
27:        L ←L_new
    return bestSub
```

Subgraph signatures When considering a new instance, one must check whether it is isomorphic to any substructure previously inserted into the new substructure list, to know if the instance should be added to an existing substructure or a new one. Checking for graph isomorphism is a costly, NP-hard operation, even with the optimized VF2 algorithm introduced by Cordella et al. [3] which was used in this work—as part of the Python package *graph-tool*. This isomorphism check is nevertheless necessary in order to correctly associate a subgraph to each of its instances in the graph.

In order to minimize the calls to the subgraph matching algorithm, apart from the use of a beam search, a method was designed to assign signatures to the subgraphs of $\mathbf{G}$, uniquely identifying the explored subgraphs. Each signature

indicates the order in which the vertices of a substructure have been discovered. It has a left side, describing a discovery path, and a right side, describing supplementary edges. Consider the following chain of instances, corresponding to the growth of an instance during subsequent iterations of the algorithm:



(13)

Throughout these expansion steps, the associated subgraph signature grows as

$$\begin{aligned}\mathtt{W},() &\Longrightarrow (\mathtt{W},(\mathtt{0},\mathtt{W})),() \Longrightarrow (\mathtt{W},(\mathtt{0},\mathtt{W}),(\mathtt{0},\mathtt{B})),() \\ &\Longrightarrow (\mathtt{W},(\mathtt{0},\mathtt{W}),(\mathtt{0},\mathtt{B}),(\mathtt{2},\mathtt{B})),() \Longrightarrow (\mathtt{W},(\mathtt{0},\mathtt{W}),(\mathtt{0},\mathtt{B}),(\mathtt{2},\mathtt{B})),(\mathtt{1},\mathtt{2})\end{aligned} \quad (14)$$

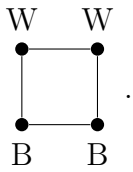
which is to say that at each expansion step

- If a new node w is added through an edge outgoing from a node v, it is added to the discovery path as a couple (n, L) with n the index of node v in the discovery path, and L the label of the new node w.
- If a new edge is added, the index of its source and target in the discovery path is added to the right side of the signature.

The use of this graph signature limits the need for isomorphism testing in the following way. At iteration $n + 1$ of the algorithm, all the substructures remaining from iteration n are extended along one edge in the graph. The subsequent discovery paths only differ on the last vertex in the discovery path, or edge in the supplementary edge list.

Because of the pruning, isomorphic substructures will often be discovered along the same discovery path throughout the algorithm: grouping the instances that have the same signature at iteration $n + 1$ already spares much of the isomorphism testing work.

Two signatures can however describe the same graph at iteration $n+1$, as in the case of a cycle



(15)

which might be described by four signatures by cyclic permutation of the labels in `(W, (0, W), (1, B), (2, B), ())`. Therefore, when a new signature s is encountered, the graph attached to this signature $\mathcal{S}$ is checked for isomorphism with the graphs attached to previously encountered signatures.

- If no isomorphism is found, signature s is stored as the *standard signature* of graph $\mathcal{S}$, and the newly discovered substructure is inserted in the list.

- If an isomorphism σ is found with a subgraph $\mathcal{S}'$, then the new signature s and the previously encountered signature s' describe the same graphs. s' has however been encountered first, and is therefore the standard signature of the graph. Therefore, the new instance nodes are reordered according to the isomorphism mapping σ, and inserted as a new instance to $\mathcal{S}'$. The signature s is stored together with the mapping σ and the signature s'; this way, if signature s is encountered again, it will not have to be checked for isomorphism anew, and will be directly identified as an alias to the standard signature s' together with the corresponding mapping σ.

This method allows to produce standard signatures restricted the subgraphs of $\mathcal{G}$ explored, and because of the pruning, only a small amount of subgraphs of $\mathcal{G}$ is encountered.

3.4.2 Results of substructure discovery

Substructure score An important element of the substructure discovery algorithm 1 is to assign a score to a substructure. As was done in [2], one solution is to focus on the ratio of the compression achieved by the substructure.

Consider the information contained in the original graph. One could consider an approximation to the minimum description length (MDL) to quantify this information, as in [2]. To simplify computation, and as was done in [10], one may also take the *size* of the graph—the sum of its number of nodes and vertices—as an approximate measure of this information.

One compresses the graph $\mathcal{G}$ by substructure $\mathcal{S}$ by replacing every instance of $\mathcal{S}$ in $\mathcal{G}$ by a single node with a new label. By knowing $\mathcal{S}$ and $\mathcal{G}|\mathcal{S}$, one can recover an approximation of $\mathcal{G}$ by replacing the single instance nodes by instances of $\mathcal{S}$. Hence, the compression score (inverse of the compression ratio) is expressed as

$$\text{score}(\mathcal{S}) = \frac{\text{size}(\mathcal{G})}{\text{size}(\mathcal{S}) + \text{size}(\mathcal{G}|\mathcal{S})} \tag{16}$$

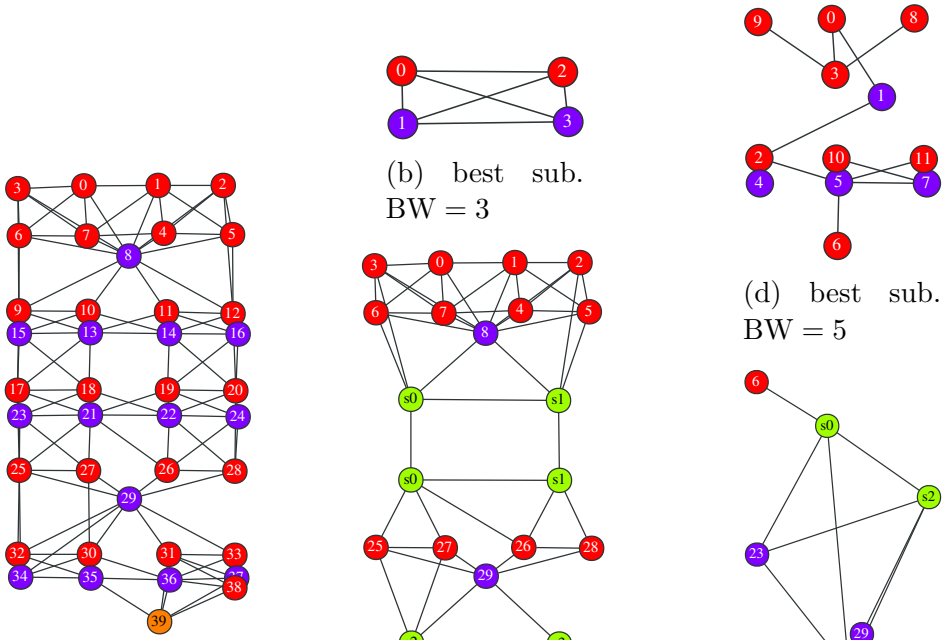
which can be computed without actually compressing the graph, using

$$\text{size}(\mathcal{G}|\mathcal{S}) = \text{size}(\mathcal{G}) - I \cdot (\text{size}(\mathcal{S}) - 1)$$

where I is the number of instances of $\mathcal{S}$ in $\mathbf{G}$. This is an approximation: it does not include the connection instructions needed to know how to connect the vertices of the instances of S to the original graph, so one could not completely recover the original graph from the knowledge of $\mathcal{S}$ and $\mathcal{G}|\mathcal{S}$. As will be discussed in Section 3.5, when building a grammar, one can add a measure of the number of needed graph productions to account for this difference.

Results Working graphs described in Section 3.2 were searched for substructures using the algorithm described earlier, implemented in the Python programming language.

The results of substructure search on the graph that was already shown in Figure 10(d) illustrates some behaviors of the algorithm. The best substructure found with a beam width of 3 differs significantly from the best substructure found with one of 5, as seen on Figure 11, and the size difference indicates that the two where found at different iteration steps.



(a) original

(b) best sub. BW = 3

(c) compressed BW = 3

(d) best sub. BW = 5

(e) compressed BW = 5

Figure 11: Best substructures found and associated graph compressions for a beam width (BW) of 3 and 5.

Figure 12 shows the evolution of the best scores of new substructures found at each iteration for various beam widths. One observes that the substructure shown on Figure 11(b) generates a local maximum in the score for every beam width bigger than 1. It is however outmatched by the substructure shown on Figure 11(d) at iteration 13 only for a beam width of 5. This shows that a low-scoring substructure at some iteration can give rise to a high-scoring substructure in subsequent growth steps. More surprising is the fact that this bigger substructure is not found using a beam width of 7, which is bigger than 5. This shows that the inclusion of more substructures can give rise to local maxima that were undetected before, and purge lower scoring substructures that would give rise to a high-scoring substructure later. Figure 13, which plots the average duration spent on each iteration as a function of the beam width, also illustrates the fact that the beam width affects the discovery process and leads to different explorations of the subgraphs of $\mathcal{G}$.

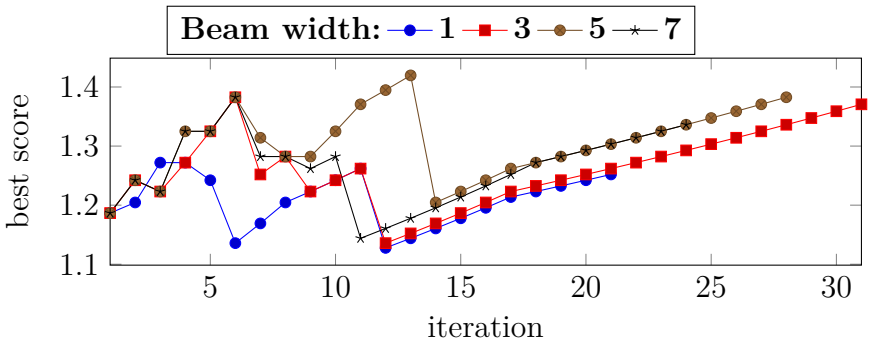


Figure 12: Best score of the new substructures found at each iteration of the algorithm, tested on the graph of Figure 11(a). Here $maxDepth = \infty$, the algorithm only stops when only no repeating substructures are discovered anymore and only singletons remain in the parent substructures list.

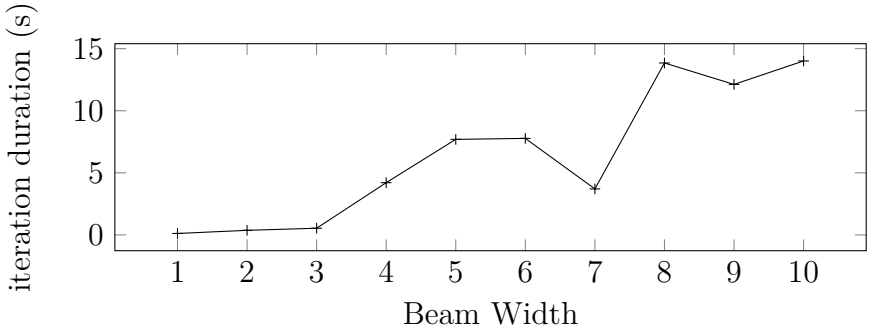
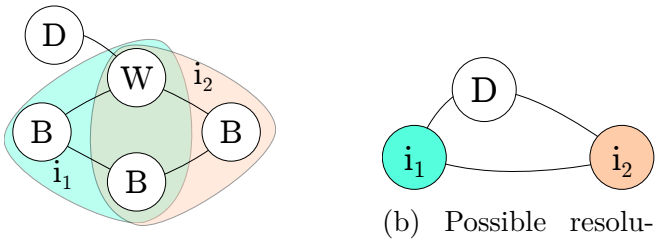


Figure 13: Average time spent on each iteration for different beam widths, using the graph of Figure 11(a) as input.

Allowing overlapping instances Previous results where obtained by searching only disjoint instances of substructures: a new encountered instance i is not added to a substructure $\mathcal{S}$ if $\mathcal{S}$ already has an instance sharing a vertex with i. However, in the edge replacement grammar learning procedure described in Section 3.3, two instances may share two vertices, forming the edge along which a new production is inferred.

This change poses problems the previous definition of the compressed graph $\mathcal{G}|\mathcal{S}$ and to the previous definition of the score of substructures. Consider for example a node with label D connected a vertex common to two substructure instances, as in Figure 14. Once the two instances have been replaced by nodes, and linked by an edge to mark the fact that they were connected in the host graph, how should D be connected to the new nodes? One solution is to connect D to both of the nodes, as in Figure 14(b). This however introduces supplementary edges in the compressed graph.

Moreover, the sharing of vertices invalidates the information theory argument behind our definition of the score of a substructure; the number of instances is badly defined, and two-vertices instances can for example be counted twice by sharing all their vertices with a second instance; Figure 14 shows the unwanted

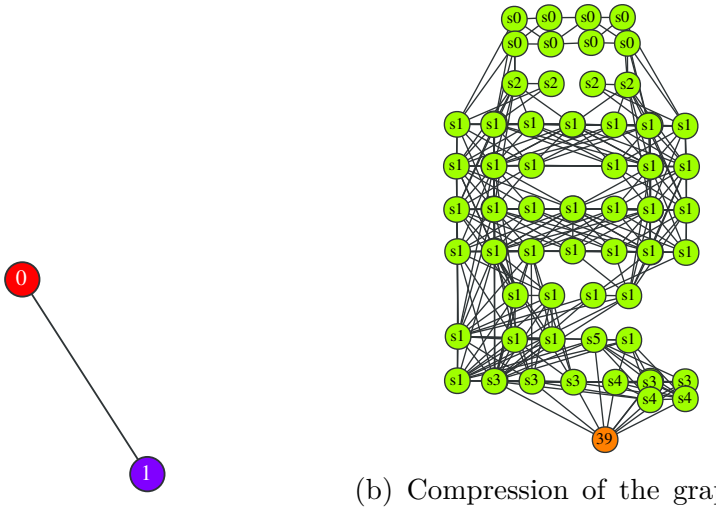


(a) Conflict

(b) Possible resolution

Figure 14: Conflict in the compression with shared vertices.

case where each instance is present two times and shares its nodes with a copy of itself.



(a) Best substructure

(b) Compression of the graph on Figure 11(a)

Figure 15: Compression with shared vertices.

The scoring function (16) might be tweaked in order to favour instances that are bigger than 2. However, this introduces complexity in the procedure and does not solve the extraneous edge problem. Therefore, in the course of this work, the search of repeating substructures has been restricted to non-overlapping instances.

Adding geometry in the edges The requirement for two instances of a substructure to be overlapped leads to an issue that has not been stated yet. Consider the simple graph represented on Figure 16 and the substructure (W)–(W)–(B). This substructure has 5 instances in the graph, but all these instances cannot be selected because of the requirement of non-overlap. Only the choice of the two yellow instances yields a repeating subgraph: the other choices select one instance in the graph only. Hence, the first chosen instance being possibly blocking other instances, even if a better packing solution existed. Since there is no way of controlling which instances will be selected by the algorithm, this

leads to a bias in the repeating substructure algorithm, the score not adequately representing the value of the subgraph.

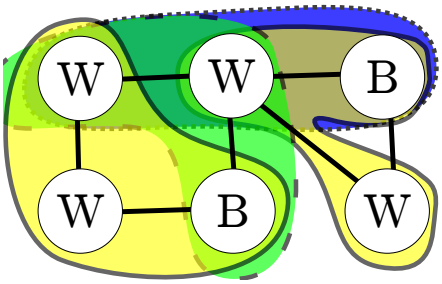


Figure 16: Overlapping instance choice problem: should the blue, the green or the yellow instances be selected?

A possible solution to this issue is to add geometric information in the edges of the graph. Indeed, one could argue that the geometric shape of the selected instances has a semantic meaning, a balcony being for example often under a window. If different geometries generate different substructures, then the choice of the instances of these substructures is made easier, and there is less risk of instances blocking other significant instances as in Figure 16.

This geometric information has been added as follows. Each edge between two components centered on c_1 and c_2 in the label image is associated to a vector $[m, t]$ with

$$m = \|\overrightarrow{c_2} - \overrightarrow{c_1}\| \tag{17}$$

the distance between the two nodes and

$$t = 1 - \left| \left(\frac{2\theta}{\pi} \bmod 2 \right) - 1 \right| \tag{18}$$

a measure in $[0, 1]$ of the angle θ of the vector $\overrightarrow{c_1 c_2}$ designed to be symmetrical in c_1 and c_2; the edges in the graph being undirected, the information contained in the edges must respect this symmetry.

In order to obtain an integer edge label, the vectors associated to all edges are clustered using a k-means algorithm into C labels ($C = 8$ in the following setup), after normalization of each component to zero mean and unit variance.

Figure 17 presents some graphs obtained after adding this geometric information in the edges.

The substructure discovery algorithm is adapted in order to distinguish two edges that do not have the same label: for instance, the labels of the edges are added to the subgraph signatures.

This new approach allows to find geometrically pertinent substructures faster, by driving the discovery process towards geometrically significant edges, and reduce the problem of choosing between overlapping instances. Table 1 shows some comparisons between substructures found with and without geometry in the edges, and compares the algorithms running times.

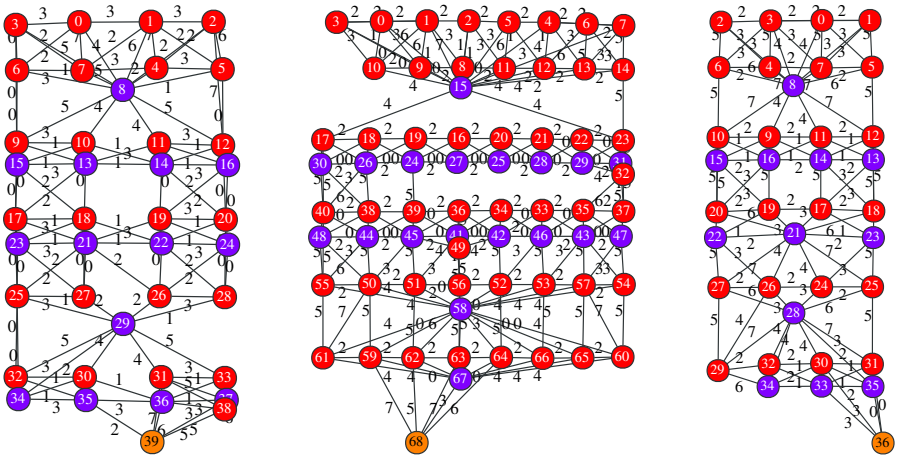


Figure 17: Adding geometric information in the edges.

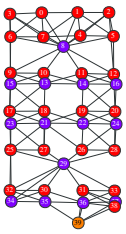
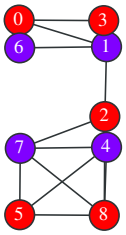
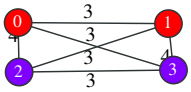
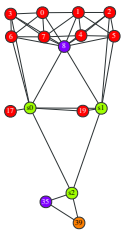
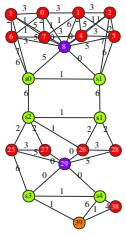
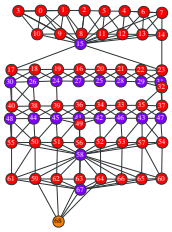
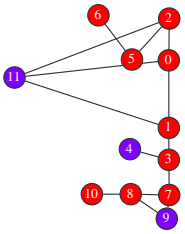
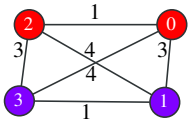
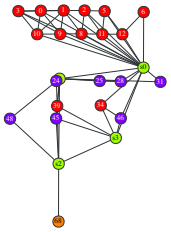
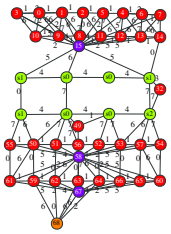
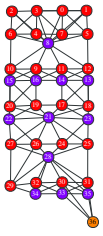
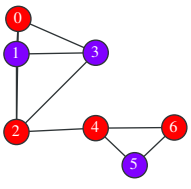
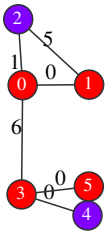
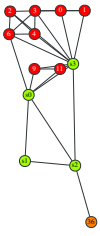
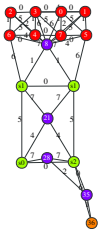
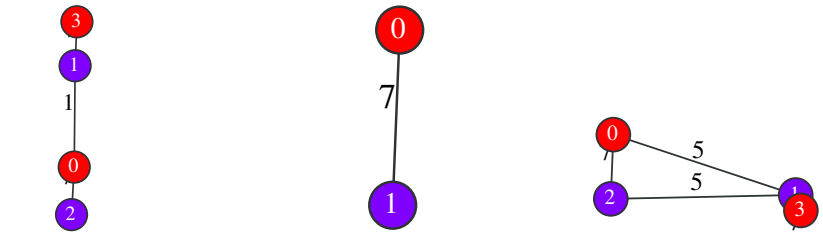
original graph	substructure		compressed graph	
	without geometry	with geometry	without geometry	with geometry
				
	20.2 s	7.8 s		
				
	46.8 s	11.4 s		
				
	22.5 s	5.0 s		

Table 1: Comparison of substructure discovery results with and without geometric information in the edges, for a beam search of width 7, and a maximum depth 15. The running time of the algorithm is shown below the substructures.

Substructure search on multiple graphs The substructure search method does not need to be adapted in order to search for repeating substructures over a whole set of graphs. Indeed, it is sufficient to apply the algorithm over a graph that is the disjoint union of all graphs in the sets, the connected components of this collection being the graphs of the set themselves.

Figure 18 show the three top-scoring substructures when running the algorithm on the whole dataset. The top-scoring substructure is a vertical chain `Balcony` – `Window` – `Balcony` – `Window`, and was found on average 17, 6 times per facade. The algorithm ran in 2m 13s (compared to 9m 44s without geometric information in the edges).



score: 1.213 score: 1.207 score: 1.200

Figure 18: Top-scoring substructures found on the whole dataset of 104 labelled images with geometric information in the edges (beam search of width 7).

3.5 Graph grammar learning

In the previous subsection, we have seen how repeating substructures of graph can be detected, and how they compresses their host graphs. The question of an adequate scoring of substructures remains, and in the case of graph grammars, the number of generated productions can be included in the measure.

We have seen that the approach is not suited for adequately detecting instances that are allowed to overlap on some number of vertices, whereas the sharing of two vertices between instances is a key ingredient of the edge replacement grammar learning method developed in [10]. Moreover, recovering the original graph in the case of graphs compressed with vertex sharing, using the conflict resolution illustrated in Figure 14(b), does not involve some simple edge replacement grammar rules, similar to those described in Section 3.3.

This leads to the conclusion that the edge grammar inference method described in [10] is not suited for inference of grammars generating the class of our working graphs. The edge recursive grammars inferred by this procedure are not expressive enough to describe the structure of our graphs of interest.

An alternative approach to graph grammar learning was developed, aiming at the inference of a NCE node replacement grammar. Such a grammar $\mathbf{G}$ replaces nonterminal nodes of a graph with new subgraphs. Each production

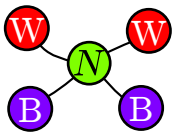
$$N \Longrightarrow \text{subgraph } \mathcal{S} \text{ with connections instructions } \mathbf{C} \tag{19}$$

comprises a nonterminal symbol N, a graph $\mathcal{S}$, and a set of connection instructions $\mathbf{C}$. A connection instruction $c \in \mathbf{C}$ has the form

$$c : v_k \longrightarrow L \tag{20}$$

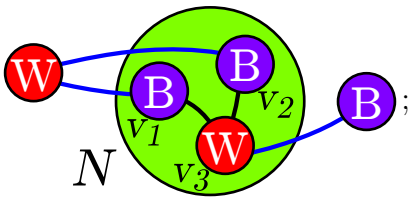
where v_k is a node of graph $\mathcal{S}$ and L a label of grammar $\mathbf{G}$, and indicates that node v_k of the new copy of $\mathcal{S}$ in the graph should be connected to all neighbours of N labelled L.

The definition and mechanism of our replacement grammar is best understood on an example. Consider a grammar $\mathbf{G_1}$comprising 2 productions, the first one

$S \Longrightarrow$ 

with no connections instructions; (21)

replacing the starting symbol S, and the second one, represented in a graphical manner as

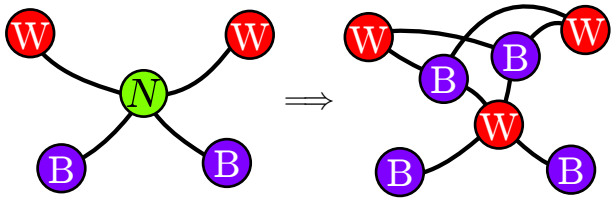


; (22)

here N is a nonterminal symbol, symbolized by the green circle; inside this circle is the new subgraph $\mathcal{S}$; the blue edges escaping this circle symbolize the production's 3 connections instructions

$$v_1 \longrightarrow W, \qquad v_2 \longrightarrow W, \qquad v_3 \longrightarrow B. \tag{23}$$

The only derivation of this grammar is

$S \Longrightarrow$ 

(24)

and shows how the neighbours of N connect to the nodes of the new subgraph according to the connection instructions represented in (22).

The method of inferring such a grammar from the substructure discovery algorithm described above is as follows:

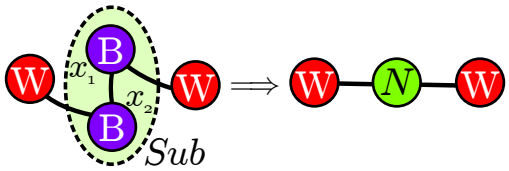
1. A frequent substructure *Sub* is found in the example graph $\mathcal{G}$;

2. The instances of *Sub* in the graph are grouped into k groups depending on how they connect to their neighbouring nodes in the graph. Each group is assigned a nonterminal symbol $S_1 \ldots S_k$ and a set of connections instructions $C_1 \ldots C_k$ describing these connections;

3. The graph is compressed by instance *Sub*.

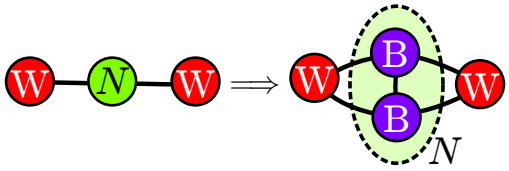
Steps 1–3 are repeated until the new instances score are under a fixed threshold and new substructures poorly compress the graph. The remaining graph $\mathcal{G}_{\text{end}}$ is changed into a new production $S \Longrightarrow \mathcal{G}_{\text{end}}$ with no connection instructions, S being the starting symbol of the grammar.

The inferred grammar does not allow to recover original graph $\mathcal{G}$ perfectly. Indeed, if one consider the following compression



(25)

then the inferred grammar rule will have connection instructions $x_1 \to W$ and $x_2 \to W$ the graph will be decompressed as



(26)

thereby adding two extraneous edges. The example graph will in any case be a subgraph of the recovered graph.

Substructure score with grammar rules The evaluation of the substructures seen in Section 1 can be adapted to take in account the grammar productions. Taking as before the size of a graph as a simple measure of the information it contains, the original graph can be exactly recovered knowing

- The substructure $\mathcal{S}$;
- The compressed graph $\mathcal{G}|\mathcal{S}$;
- For each inferred production p_i, the connection instructions C_i;
- The set of extraneous edges $\overline{E}$ which one should remove from the decompressed graph.

Therefore, a sensible score that can be given to a substructure is

$$\text{score}(\mathcal{S}) = \frac{\text{size}(\mathcal{G})}{\text{size}(\mathcal{S}) + \text{size}(\mathcal{G}|\mathcal{S}) + \sum_{p_i} |C_i| + |\overline{E}|}. \tag{27}$$

Again one has

$$\text{size}(\mathcal{G}|\mathcal{S}) = \text{size}(\mathcal{G}) - I \cdot (\text{size}(\mathcal{S}) - 1)$$

and by grouping the instances connections and inspecting the outgoing edges during substructure search, score (27) can be computed without actually compressing the graph, which is important for efficiency.

3.5.1 Results

Compression and decompression: first grammar layer Table 2 presents the substructure search, compression and decompression results obtained on 3 input labels. Interestingly, the same substructure is found on the three input labels.

Multiple compression levels Figure 19 shows the compression chain of an input graph: the input graph is compressed until the compressed graph contains no repeating substructure. One observes that score (27) favours small substructures of 2 or 3 nodes in the graphs of the compression chain. This measure could be altered to favor bigger substructures to suit a particular Computer Vision task.

Multiple decompressions The introduction of extraneous edges at a bottom layer of the graph compression chain affects all layers above. Therefore, a reconstruction of the original graph is tractable only after a few compressions, as is shown on Figure 20.

Working on multiple images As can be noticed in previous results, the grammars generated from single labels share similarities, in the nature and order of discovery of their substructures. A method of working with several graphs together would help describing a general grammar of facades. However, the method described earlier, consisting in concatenating the input graphs, is not suited for grammar productions discovery: here the algorithm would learn how to reconstruct a set of facades and not a single facade instance.

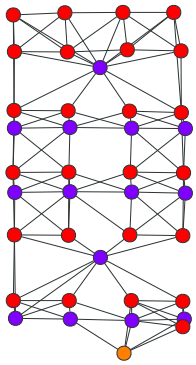
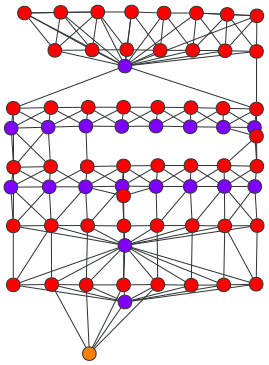
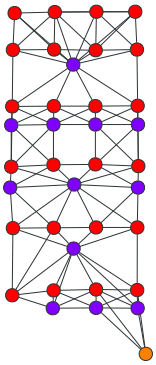
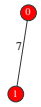
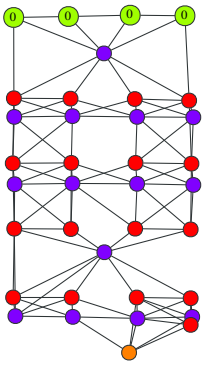
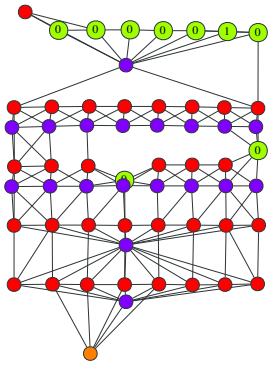
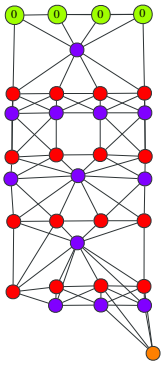
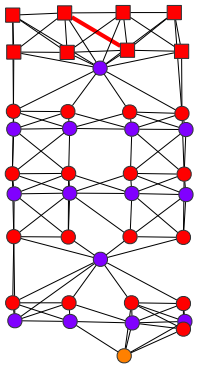
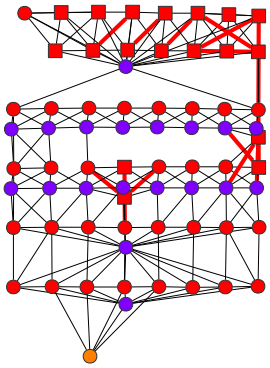
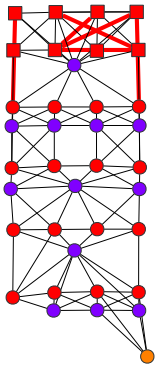
original			
best subst.			
score	0.993	0.971	0.967
compressed			
instance kinds	1	2	1
decompressed			
extraneous edges	1	14	6

Table 2: Compressions/decompressions of 3 input labels. Substructures were searched separately on input labels. Reconstructed nodes are square shaped, and extraneous edges, introduced by reconstruction, are thick and red.

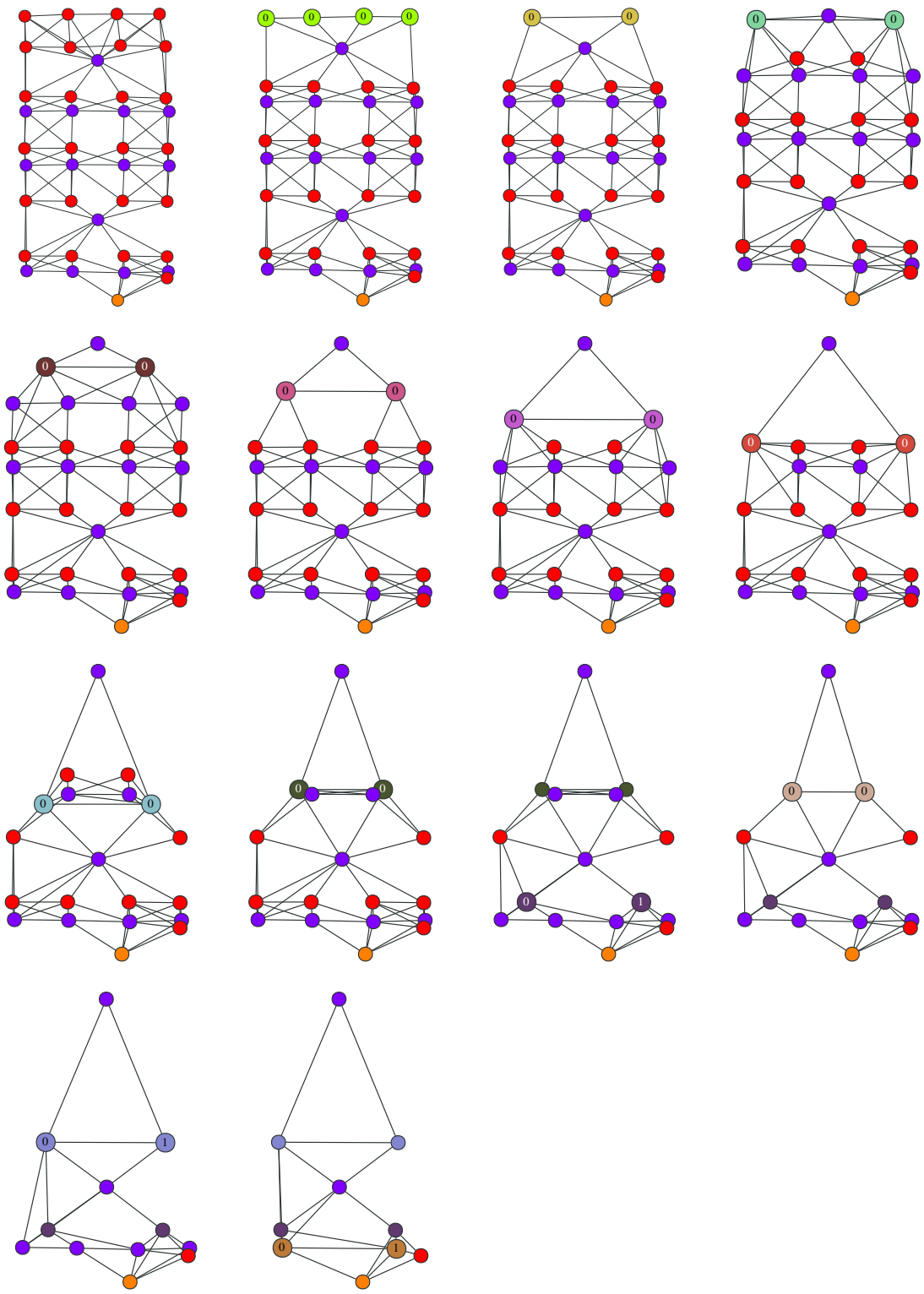


Figure 19: Repeated compression of an input graph. Top Left: input graph. Bottom Right: final compressed graph.

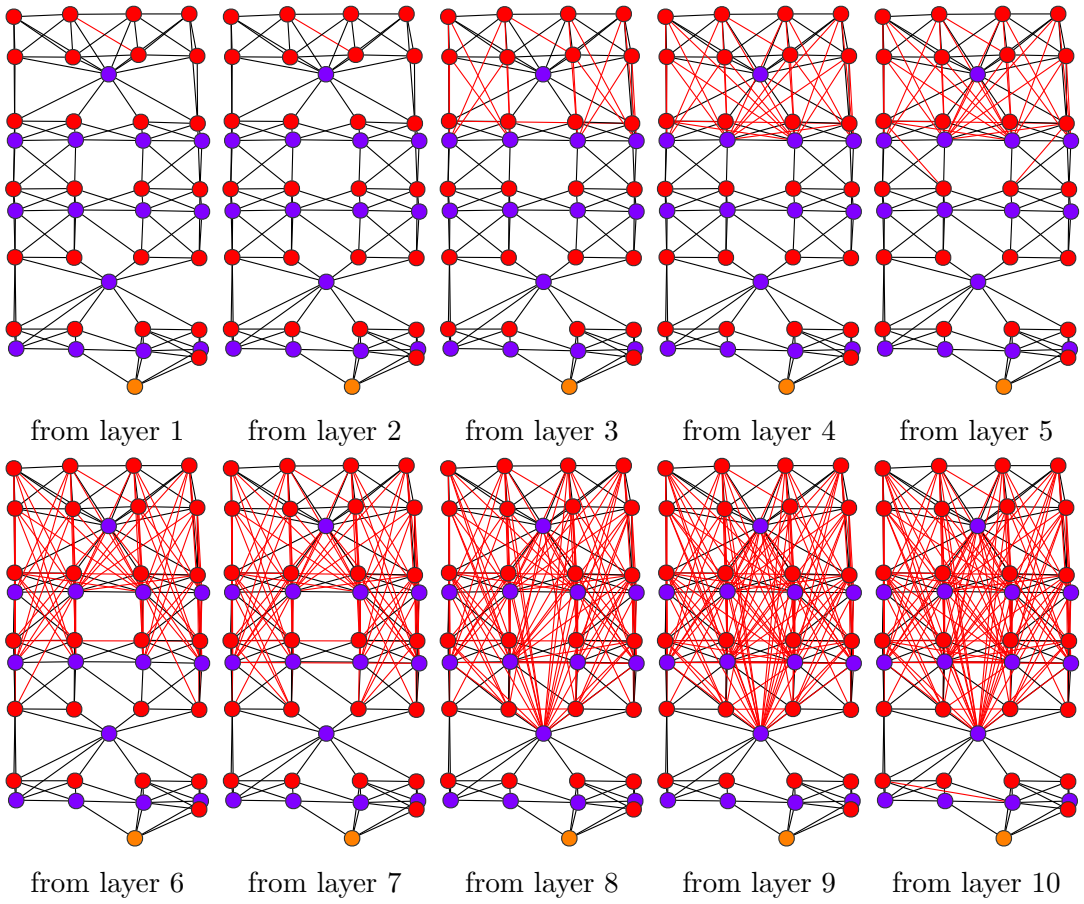


Figure 20: Reconstruction of the original graph, starting from different grammar layers. The extraneous edges (in red) become important quickly.

Conclusions and Perspectives

The widespread use of low-cost computing power in electronics has enabled the development of high-performance Computer Vision systems relying on high-dimensional low level image features and running on consumer devices. In order to introduce semantic information in images and handle the high compositional complexity of natural scenes, it seems however necessary to go beyond low level representations and introduce object-level models on top of image features. These models should have sufficient degrees of freedom to account for the high variability of appearances and object arrangements, while staying simple enough to be learned and used in a tractable way.

Grammatical models are one of the promising possibilities. Their previous use in Computer Vision does however suffer from limitations of the grammar formalism, engendering tree-like representations without really making use of the full processing power of grammars, which allows repetition of rules. The graph grammar formalism is richer and able to describe whole classes of graphs, mathematical objects that are omnipresent in Computer Vision today. However, there has been little work on graph grammar inference, which is a key for their adoption as object models.

This work introduces methods for the introduction and learning of graph grammar formalisms in Computer Vision. Its main contribution is an efficient Computer Vision-oriented substructure discovery algorithm, and the introduction of chained partially reversible compressions of graphs. The use of this algorithm to assist and guide a Computer Vision task is the object of further work. The first attempt of graph grammar generation falls short of inferring a grammar describing classes of images, and the extraneous edges introduced by the lossy substructure compression hinders the fidelity of the grammatical representation of image labels. Moreover, the score given to a substructure should be adapted to suit a particular use of the model, as the generic scoring function given in this work favors small grammar substructures, which seems unsatisfactory to account for high level graphs structure.

However, the similarity of the observed patterns between compressions of different labels suggests that chained substructure discovery can be used as a signature of high level structure of objects. Instead of trying to reconstruct a graph with perfect fidelity using a graph grammar and introducing extraneous edges, one possibility would be to develop inexact grammar production matching using inexact graph matching techniques and allow for alterations of the original graph for it to suit better the expressivity of a graph grammar model.

By answering some questions on the introduction of graph grammars in Computer Vision, this work opens up many others, among which the implementation of a graph grammar model in an actual object detector.

References

[1] N. Chomsky. Three models for the description of language. *Information Theory, IRE Transactions on*, 2(3):113–124, 1956.

[2] D. J. Cook and L. B. Holder. Substructure discovery using minimum description length and background knowledge. *arXiv preprint cs/9402102*, 1994.

[3] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. An improved algorithm for matching large graphs. In *3rd IAPR-TC15 workshop on graph-based representations in pattern recognition*, pages 149–159, 2001.

[4] M. Everingham, L. Van Gool, C. Williams, J. Winn, and A. Zisserman. The pascal visual object classes challenge 2007 (voc 2007) results (2007). In *URL http://www. pascal-network. org/challenges/VOC/voc2007/workshop/index. html*, 2008.

[5] P. Felzenszwalb, D. McAllester, and D. Ramanan. A discriminatively trained, multiscale, deformable part model. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE, 2008.

[6] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan. Object detection with discriminatively trained part-based models. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 32(9):1627–1645, 2010.

[7] P. F. Felzenszwalb and D. McAllester. Object detection grammars. In *ICCV Workshops*, page 691, 2011.

[8] L. Fürst, M. Mernik, and V. Mahnič. Graph grammar induction as a parser-controlled heuristic search process. In *Applications of Graph Transformations with Industrial Relevance*, pages 121–136. Springer, 2012.

[9] J. P. Kukluk, L. B. Holder, and D. J. Cook. Inference of node replacement recursive graph grammars. In *SDM*, pages 544–548. SIAM, 2006.

[10] J. P. Kukluk, L. B. Holder, and D. J. Cook. Inference of edge replacement graph grammars. *International Journal on Artificial Intelligence Tools*, 17(03):539–554, 2008.

[11] A. Lindenmayer. Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of theoretical biology*, 18(3):280–299, 1968.

[12] A. L. P. Prusinkiewicz, A. Lindenmayer, J. S. Hanan, F. D. Fracchia, and D. Fowler. The algorithmic beauty of plants with. 1990.

[13] G. Rozenberg and H. Ehrig. *Handbook of graph grammars and computing by graph transformation*, volume 1. World scientific Singapore, 1999.

[14] G. Stiny. Spatial relations and grammars. *Environment and Planning B: Planning and Design*, 9(1):113–114, 1982.

[15] O. Teboul. École Centrale Paris facades database, 2010.

[16] O. Teboul, I. Kokkinos, L. Simon, P. Koutsourakis, and N. Paragios. Shape grammar parsing via reinforcement learning. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 2273–2280. IEEE, 2011.

[17] L. G. Valiant. General context-free recognition in less than cubic time. *Journal of computer and system sciences*, 10(2):308–315, 1975.

[18] J. Weissenberg, H. Riemenschneider, M. Prasad, and L. Van Gool. Is there a procedural logic to architecture? In *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*, pages 185–192. IEEE, 2013.

[19] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky. *Instant architecture*, volume 22. ACM, 2003.