Technical Report

Pattern Recognition and Image Processing Group
Institute of Visual Computing and Human-Centered Technology
TU Wien
Favoritenstrasse 9-11/193-03
A-1040 Vienna AUSTRIA
Phone: +43 (1) 58801 - 18661
Fax: +43 (1) 58801 - 18697
E-mail: l.boccia@pm.me
URL: http://www.prip.tuwien.ac.at/

PRIP-TR-152 May 4, 2022

# Efficient Contraction Kernel Selection with a Knight's Move[1]

*Luca Boccia and Walter G. Kropatsch*

**Abstract**

Irregular image pyramids built with combinatorial map representations of graphs (Combinatorial Pyramids) enable the usage of topological information for both down-sampling and up-sampling, used for compactness and for surface reconstruction respectively. The hierarchy and the complexity of the image pyramid are mainly controlled by the choice of a contraction kernel. In this work we propose a formal grammar for selecting contraction kernels for plateau regions in binary and multi-label images. Our method enables parallel computing through selection of independent edges to contract, and efficiency is achieved by finding a good balance between the number of contraction and simplification operations, needed after the application of the kernel. We show that our method achieves regularity and produces pyramids of height similar to regular pyramid schemes on plateau regions. Finally, we present possible solutions to generalize our approach to design a Connected Component Labeling algorithm that is both efficient and correct.

[1]Part of the work is related to the WWTF project LS19-013 "Water's Gateway to Heaven". The support is gratefully acknowledged.

# Acknowledgements

I would like to extend my deepest gratitude to prof. Walter Kropatsch without whom it would have not been possible for me and my colleagues to be working at PRIP in the first place. His mentoring shaped this work and my views of the future. I would like to recognize the invaluable assistance of Darshan, for proposing the idea behind this work, for his tireless explanations on the subject, for our fiery discussions, and for our friendship. I would like to thank Jiří and Majid, for the questions they asked during the multiple presentations of this work, to which they already had an answer, but I did not at the time. I am very grateful to Carmine C. and Carmine N. for all the (non-) productive confrontations in our shared kitchen. Finally, I should thank prof. Mario Vento for believing in this experience and for his support from its beginning until the end.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Introduction

This work is an extension of [5], master thesis of Boccia.

Binary and multi-label images are largely used for tasks like image processing and image segmentation. These images are composed of large areas of the same color value, called plateau regions, which carry the information necessary to the task at hand.

Because of these large plateaus, binary and multi-label images lack information about structure, which is instead usually found in textures or in critical points and their connections [3]. This kind of information communicates topological information efficiently, and its absence makes algorithms for contraction kernel selection use random strategies to choose the set of edges to contract and their surviving vertices [3]. In [2], an objective function has been proposed to optimize contraction kernel selection, but, on plateaus, all edges have the same contrast, and thus the same priority. In this case, the choice of a contraction kernel is arbitrary. The definition of a strategy for the selection of contraction kernels on plateau regions will make such choices non-arbitrary and optimal.

The goal of this work is to adapt the pyramid scheme proposed in [2] to combinatorial pyramids. We show that, under certain assumptions, the scheme makes it possible to generate contraction kernels that are applicable in parallel in two steps. To allow parallel computing, a contraction operation on an edge must not be dependent on other operations that will be carried out at the same time, taking into account the properties of the data structure that is being used [2]. We show that this scheme ensures the absence of dependencies between selected edges, and we propose a fallback strategy for the cases in which the necessary assumptions do not hold.

Application of a contraction kernel to a graph will also result in redundant information, namely in double edges and empty self-loops. Removal of these redundant edges adds complexity, and, like with contraction operations, dependencies have to be taken into account when using a parallel strategy for it.

We show that this scheme produces new levels on which it is possible to apply the same scheme with the same assumptions. This adds predictability to the output of a contraction kernel application, which makes it possible to approximate levels of the pyramid with information gathered from adjacent levels. This could lead the way to combinatorial pyramid compression, where the state of the art for combinatorial pyramids storage has linear space complexity with respect to the base level [18].

Using a precise scheme means reaching regularity in the number of levels of the pyramid, similarly to regular image pyramids where a fixed reduction factor in regular image pyramids generates pyramids of height of logarithmic complexity with respect to the dimensions of the image. In irregular image pyramids, the height of the pyramid is governed by the choice of the contraction kernels. With our method, plateau regions are contracted with an approximately regular reduction factor, and the irregular pyramid achieves properties of complexity and height comparable to regular image pyramids.

## Structure of the Work

In Chapter 1 we introduce the most important concepts that are needed to grasp the novelties introduced in this work. We introduce graph representation of images, combinatorial maps, irregular pyramids and formal grammars, all of which are instruments at the heart of this work.

In Chapter 2 we define the knight's move scheme, and we prove some of its properties. We also propose a grammar to traverse the graph, which will be used as a model for the implementation.

In Chapter 3 we describe the implementation of a Connected Component Labeling algorithm based on the knight's move scheme. We present the assumptions and the decisions that have been made to solve practical problems that arise when these assumptions do not hold. We also describe the most relevant techniques that have been used for parallel computing.

In Chapter 4 we show the effectiveness of our approach through execution of the forementioned algorithm on benchmarks, and we also show some of its faults and drawbacks. Finally, we analyze different possible solutions to these problems for further developments.

Chapter 1

# Images, Graphs and Formal Grammars

In order to understand the goals of this work and the instruments that we use to achieve these goals, in this chapter we are presenting the main concepts, definitions and notations that we use throughout the whole work. In Section 1.1 we introduce the definition of graphs built on images and other related concepts. In Section 1.2 we define combinatorial maps as a representation of graphs, which will be used extensively in this work. In Section 1.3 we explore the literature about image pyramids, schemes to build them and how irregular pyramids become the natural successor to these. In Section 1.4 we put all together to define irregular image pyramids that are based on combinatorial map representations of graphs, called combinatorial pyramids. Finally, in Section 1.5 we deal with formal grammars to the extent that is required in this work.

## 1.1 Graph Representation of Images

A discrete $m \times n$ image $P$, where each pixel $p \in P$ is associated to a color encoding vector, can be represented as a planar 4-neighborhood graph $G(V, E)$, where a vertex $v \in V$ represents a pixel, and its edges $e \in E$ represent the relationship of the pixels with their 4 adjacent neighbors. If we consider a pixel as the smallest region of the graph, this can be interpreted as a Region Adjacency Graph (RAG) [3]. The resulting graph is an $m \times n$ grid graph.

**Definition 1.** A two-dimensional *grid graph* is an $m \times n$ lattice graph that is the graph Cartesian product $P_m \square P_n$, where $P_m$ and $P_n$ are path graphs of $m$ and $n$ vertices, respectively.

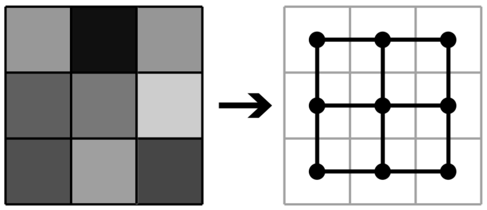


Figure 1.1: Example of RAG Built on an Image.

Each vertex will be associated to a weight derived from each pixel's color encoding vector. Without loss of generality, for the rest of this work we will refer to the gray value of the pixels, denoted as $g(p)$, as the one to be associated to each vertex, such that $g(v) = g(p)$. Thus, we assume that such weights respect the same order properties of the set of real numbers $\mathbb{R}$.

The weight of an edge $e = (v, w) \in E$ will be the contrast between its endpoints $c(e) = |g(v) - g(w)|$. The orientation of an edge is *directed from vertex $v$ to vertex $w$* if $g(v) > g(w)$. The edge is not oriented if $g(v) = g(w)$.

**Definition 2.** A vertex $v \in V$ is a *boundary vertex* if its associated pixel $p \in P$ is on the border of the image. Otherwise, it is called *inner vertex* (or inside vertex).

**Definition 3.** An edge $e \in E$ is a *boundary edge* if its endpoints are boundary vertices. Otherwise, it is called *inner edge* (or inside edge).

*Remark.* Let $E_b$ be the set of boundary edges, $E_i$ the set of inner edges. It holds that $E_b \cup E_i = E$ and $E_b \cap E_i = \emptyset$. Analogously, the same properties hold for the set of inner and boundary vertices.

**Definition 4.** A *plateau region* is a connected subgraph $G'$ of $G$ having the same gray value for all the vertices, that is $\forall v, w \in V_{G'}, g(v) = g(w)$.

## 1.2 Combinatorial Maps

Combinatorial maps were first introduced in 1960 by Edmonds [12].

**Definition 5.** A *combinatorial map* is a triplet $C = (\mathcal{D}, \alpha, \sigma)$, where:

- $\mathcal{D}$ is a finite set of darts,
- $\alpha$ is an involution on the set $\mathcal{D}$, and
- $\sigma$ is a permutation on the set $\mathcal{D}$.

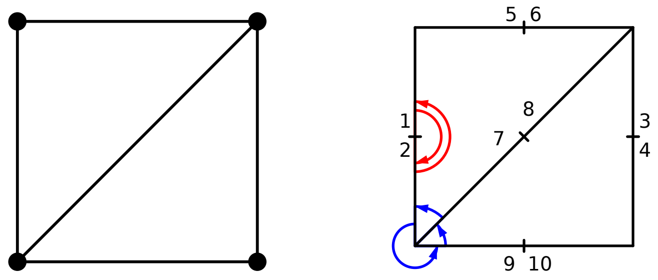


Figure 1.2: Example of a Planar Graph (left) Encoded as a Combinatorial Map (right). The numbered segments represent darts; darts linked by the involution $\alpha$ (in red) are separated by a small bar; blue arrows represent the permutation $\sigma$.

A planar graph can be represented with a combinatorial map (see Fig. 1.2). Each edge is represented as two darts $d_1, d_2 \in \mathcal{D}$ connected by the involution $\alpha$, such that $\alpha(d_1) = d_2$ and $\alpha(d_2) = d_1$. Trivially, the relationship $\alpha(\alpha(d)) = d$ holds for each dart, from which we derive the definition of (alpha)-*orbit* of $d$ as the sequence of $d$ and $\alpha(d)$, written as $\alpha^*(d)$, with which we may refer to an edge of the graph.

Each dart implicitly represents the vertex to which it is incident. The permutation $\sigma$ connects a dart $d_1$ to the following dart $d_2$ incident to the same vertex in clockwise or counter-clockwise direction. Analogously to the (alpha)-*orbit*, the sequence of all darts incident to the same vertex starting from dart $d$ in (counter-) clockwise direction is called (sigma)-*orbit* of $d$, and it is written as $\sigma^*(d)$. The direction is arbitrary and fixed, and for the rest of this work it will be assumed as counter-clockwise.

A combinatorial map implicitly encodes a dual graph, which can be constructed using a combination of $\alpha$ and $\sigma$. The encoding of the edges for the dual graph is the same as the primal, namely using the $\alpha$ involution, and the permutation $\varphi$ of the dual graph has the opposite direction of $\sigma$, and it is defined as $\varphi = \alpha \circ \sigma$ or $\varphi = \sigma \circ \alpha$. Thus, it holds that

$$C = (\mathcal{D}, \alpha, \sigma) \iff \overline{C} = (\mathcal{D}, \alpha, \varphi),$$

where $\overline{C}$ is the dual of $C$. In the primal graph, $\varphi^*(d)$ returns all the darts turning around a face.

**Property 1.** If the primal combinatorial map is connected, its dual is connected too. [6]

Special types of edges are described by Brun and Kropatsch in [6] as follows:

**Definition 6.** An edge $\alpha^*(d)$ is called an *empty self-loop* if and only if $\sigma(d) = \alpha(d)$.

**Definition 7.** An edge $\alpha^*(d)$ is called a (non-empty) *self-loop* if and only if $\alpha(d) = \sigma^*(d)$.

**Definition 8.** A dart $d$ is called a *pendant dart* if and only if $\sigma(d) = d$. The vertex $\sigma^*(d) = (d)$ is called a *pendant vertex*.

**Definition 9.** An edge $\alpha^*(d)$ is called *pendant edge* if and only if $d$ or $\alpha(d)$ are pendant darts.

**Definition 10.** An edge $\alpha^*(d)$ is called a bridge iff $\alpha(d) \in \varphi^*(d)$.

These special edges are also related between the primal and dual graph [7]:

| Primal $C = (\mathcal{D}, \alpha, \sigma)$ | Dual $\overline{C} = (\mathcal{D}, \alpha, \varphi)$ |
|---|---|
| self-loop | bridge |
| bridge | self-loop |
| empty self-loop | pendant dart |
| pendant dart | empty self-loop |

## 1.3 Image Pyramids

An image pyramid is a multi-level representation of an image, in which a hierarchy of levels is defined, and each level is related to the previous and the next. Regular image pyramids have been introduced in 1981 by the authors in [10] as a stack of images of decreasing resolution, where each image is a level of the pyramid; the lowest level is the original image and the highest corresponds to the weighted average of the original [7].

Since their introduction in 1981, image pyramids have been used in many applications, such as data compression, multiscale texture analysis, shape analysis, motion analysis, image blending, or multiscale object detection. [1] and [17] are examples of surveys which describe these and other applications of image pyramids.

Classic examples of regular image pyramids are Gaussian and Laplacian pyramids (see Fig. 1.3a). Gaussian pyramids are defined by down-sampling the input image on each level by picking even rows and columns, thus reducing its resolution of a factor of 4 per level. The down-sampling operation is preceded by a Gaussian blur filter application. Each level needs to be stored

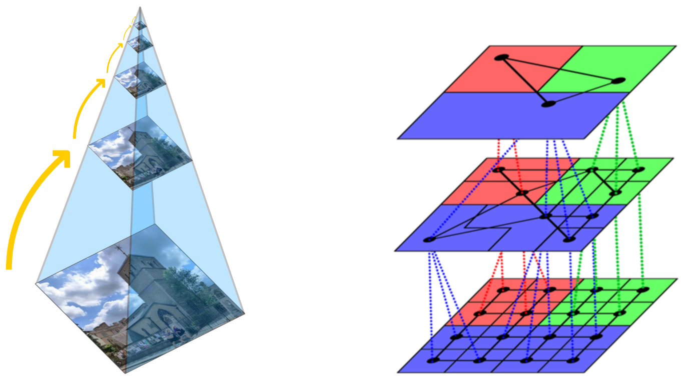


Figure 1.3: Examples of Image Pyramids. (a) represents a regular image pyramid, and (b) represents an irregular image pyramid.

to not lose information. By contrast, Laplacian pyramids only store the last level and the residual images of every other level, namely, the difference between the level itself and the same after the blur filter is applied. Laplacian pyramids have been successfully used as a compact representation of images, as described in [9], given that residual images contain less information than their counterparts in a Gaussian pyramid.

### 1.3.1 Regular Pyramid Schemes

In regular pyramids, the *reduction window* relates each pixel of a level to a set of pixels of the level below; this set is called *receptive field*. This relationship can be described as a parent-child relationship [7]. The shape of the reduction window is fixed. The ratio between the size of a level and its successor is also fixed, and it is called *reduction factor* of the pyramid.

We can denote pyramid schemes by considering the shape of the reduction window in number of pixels that merge into one, and the consequent reduction factor. A pyramid scheme that merges four pixels into one equivalent in the center, called quad-tree pyramid, has a $2 \times 2$ mask and reduction factor of 4, and will be denoted as $2 \times 2/4$-Pyramid (see Fig. 1.4a). An example of this scheme are Gaussian and Laplacian pyramids.

The formula $2 \times 2/4$ has quotient 1, which expresses the non-overlapping nature of the mask. Pyramid schemes where not all pixels are used during reduction have a quotient $< 1$, and quotients $> 1$ indicate the use of overlapping masks, where pixels are used in more than one reduction operations.

In order to represent non-rectangular masks, we may add or subtract a number of pixels from a rectangular mask, mostly in a center symmetric way. Examples of all kinds of pyramid scheme can be found in Fig. 1.4.

We require that the representative pixels $(x_{n+1}, y_{n+1})$ of the next level of the pyramid are either on a pixel location of the level below (with integer coordinates) or at dual inter-pixel locations:

$$(x_{n+1}, y_{n+1}) = (x_n, y_n) + \delta, \qquad \text{with } \delta \in \underbrace{\left\{0, \frac{1}{2}\right\}^2}_{\text{the Cartesian product of the set by itself}}.$$

In addition, the new unit vectors $(dx_1, dy_1), (dx_2, dy_2)$ must be orthogonal, that is

$$dx_1 dx_2 + dy_1 dy_2 = 0,$$

and have the same (real valued) length

$$dx_1^2 + dy_1^2 = dx_2^2 + dy_2^2.$$

### 1.3.2 Irregular Pyramids

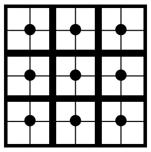
Regular image pyramids have a rigid and fixed structure, which introduces several drawbacks like the shift-dependence problem and a limited number of regions encoded at any given level [4]. Irregular pyramids have been introduced by the authors in [15] to overcome these negative properties, while keeping the main advantages. These are defined as a stack of successively reduced graphs, where a vertex in a given level of the pyramid, called *surviving vertex*, is mapped to a set of vertices in the level below, called *non-surviving vertices* [15], similarly to pixel relationships in regular pyramids (see Fig. 1.3b).

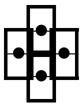
If we refer to graph representations of images defined in Section 1.1, we can define irregular image pyramids with analogous purpose and properties of regular image pyramids. Thus, we may call these *irregular image pyramids*.

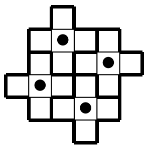
The process of reduction of said graphs is described by Kropatsch in [13] as a scheme for dual graphs. In this scheme, given a graph, we can define a contraction kernel.

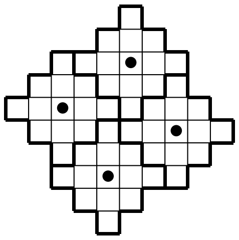
**Definition 11.** Let $G = (V, E)$ be a graph, then a *contraction kernel* $\mathcal{K}$ of $G$ is defined as a set $\mathcal{S} \subset V$ of surviving vertices and a set $\mathcal{N} \subseteq E$ of non-surviving edges such that:

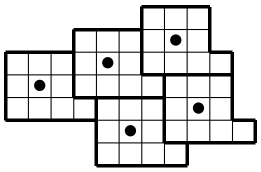
- $(V, \mathcal{N})$ is a spanning forest of $G$,

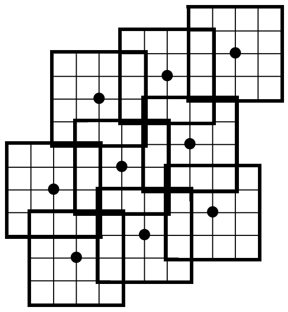

(a) $2 \times 2/4$


(b) $1 \times 2/2$


(c) $(3 \times 3 - 4)/5$


(d) $(5 \times 5 - 12)/13$


(e) $(3 \times 3 + 1)/10$


(f) $4 \times 4/10$

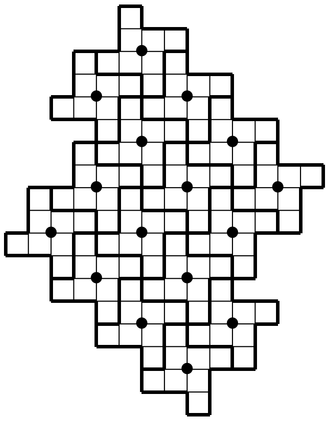

(g) $(4 \times 4 - 8)/8$

Figure 1.4: Regular Pyramid Schemes. The receptive field is denoted by the thicker line; the representative pixel position is denoted by the bullet (●).

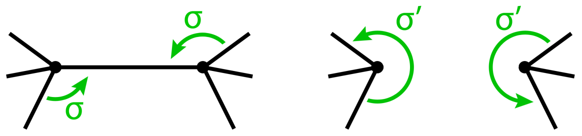


Figure 1.5: Removal Operation on a Combinatorial Map.

- Each tree of $(V, \mathcal{N})$ is rooted in a surviving vertex $v \in \mathcal{S}$.

Application of a contraction kernel consists in (1) a contraction operation in the primal graph, called *dual edge contraction*, which contracts non-survivor edges towards their surviving vertex, and (2) a contraction operation in the dual graph, called *dual face contraction*, which simplifies the graph by removing multiple edges and empty self-loops.

## 1.4 Combinatorial Pyramids

If we encode an image graph using a combinatorial map representation as in Section 1.2, an irregular pyramid can be constructed from a base combinatorial map by repeatedly applying the reduction scheme seen in Section 1.3.2 by means of edge contraction and removal operations. Such irregular pyramids we call *combinatorial pyramids*. We will show that these operations, applied on a combinatorial map, consist just of changes in the permutation $\sigma$ of said map.

The removal operation can be seen as the skipping of a dart in the (sigma)-*orbit* of the endpoints of the removed edge (see Fig. 1.5).

**Definition 12.** Let $C = (\mathcal{D}, \alpha, \sigma)$ be a combinatorial map and let $d \in \mathcal{D}$ be a dart of $C$ such that $d$ is neither a bridge nor a self-loop. The removal of the edge $\alpha^*(d)$ from $C$ produces the combinatorial map

$$C' = (\mathcal{D} \setminus \alpha^*(d), \alpha, \sigma')$$

where

$$\sigma'(d') = \begin{cases} \sigma(d') & \text{if } d' \in \mathcal{D} \setminus \sigma^{-1}\left(\alpha^*(d)\right) \\ \sigma(d) & \text{if } d' = \sigma^{-1}\left(d\right) \\ \sigma(\alpha(d)) & \text{if } d' = \sigma^{-1}\left(\alpha(d)\right) \end{cases} .$$

Similarly, the contraction operation consists of merging the (sigma)-*orbit* of the endpoints of the contracted edge (see Fig. 1.6).

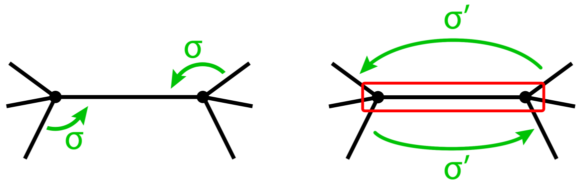


Figure 1.6: Contraction Operation on a Combinatorial Map.

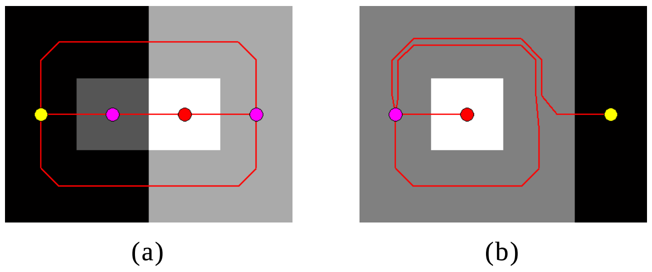


Figure 1.7: Example of the Topological Preservation Properties of Combinatorial Pyramids.

**Definition 13.** Let $C = (\mathcal{D}, \alpha, \sigma)$ be a combinatorial map and let $d \in \mathcal{D}$ be a dart of $C$ such that $d$ is neither a pendant dart nor a self-loop. The contraction of the edge $\alpha^*(d)$ from $C$ produces the combinatorial map

$$C' = (\mathcal{D} \setminus \alpha^*(d), \alpha, \sigma')$$

where

$$\sigma'(d') = \begin{cases} \sigma(d') & \text{if } d' \in \mathcal{D} \setminus \sigma^{-1}(\alpha^*(d)) \\ \sigma(\alpha(d)) & \text{if } d' = \sigma^{-1}(d) \\ \sigma(d) & \text{if } d' = \sigma^{-1}(\alpha(d)) \end{cases}.$$

Some types of special edges are excluded from the contraction because we want to preserve the connectivity of the image itself. For example, removing a bridge in the primal graph would destroy connectivity by definition, which translates to the contraction of a self-loop in the dual graph.

A combinatorial pyramid has a few advantages over dual graph pyramids. First, it is trivial to see that there is a more concise representation of both the primal and dual graph, given that the latter is implicitly encoded. Second, combinatorial pyramids preserve topological information thanks to combinatorial map properties [6]. In combinatorial maps it is possible to distinguish between empty and non-empty self-loops, otherwise indistinguishable in simple graphs, which leads to the differentiation of adjacency and inclusion relationships between regions at higher levels of the pyramid (see Fig. 1.7).

It is also worth noting that Torres and Kropatsch in [18] propose a canonical representation with which combinatorial pyramids can be stored with exactly the same space as the base level.

## 1.5 Formal Grammars

The following section is derived from [14].

Given a set of symbols $\Sigma$, called an *alphabet*, we can construct finite sequences of symbols called *strings*, e.g., $w = aabbbaa$ with $a, b \in \Sigma$ is a string.

We can define the *concatenation* operation of two strings

$$v = a_0 \dots a_n,$$
$$w = b_0 \dots b_m$$

as

$$vw = a_0 \dots a_n b_0 \dots b_m.$$

The empty string is denoted with $\varepsilon$, and $\forall w$ it holds

$$\varepsilon w = w\varepsilon = w.$$

With $w^n$ we denote the string obtained by concatenating $w$ $n$ times, and we define $w^0 = \varepsilon$. If $\Sigma$ is an alphabet, the *Kleene star* (or Kleene closure) of $\Sigma$, denoted as $\Sigma^*$, is the set of all finite strings obtained by concatenating zero or more symbols from $\Sigma$. If $a$ is a symbol,

$$a^* = \{\varepsilon, a, aa, aaa, \dots\},$$

a special case we will use frequently.

**Definition 14.** A *grammar* is a quadruple

$$G = (V_T, V_N, S, P)$$

where $V_T$ is the set of *terminal symbols*, $V_N$ is the set of *non-terminal symbols*, $S \in V_N$ is the *start symbol* and $P$ is a set of *production rules.* We assume $V_T \cap V_N = \emptyset$.

Production rules in $P$ are in the form

$$x \to y$$

where $x \in (V_T \cup V_N)^* \, V_N \, (V_T \cup V_N)^*$, i.e. $x$ is a string containing at least a symbol from the non-terminal set, and $y \in (V_T \cup V_N)^*$. A production rule $x \to y$ is applicable to a string $w = uxv$ by substituting $x$ in $w$ with $y$, obtaining $z = uyv$. We say that $w$ *derives* $z$, or that $z$ is derived from $w$, and it is written as

$$w \Rightarrow z.$$

If

$$w_0 \Rightarrow w_1 \Rightarrow \cdots \Rightarrow w_n$$

we say that $w_0$ derives $w_n$, and we write

$$w_0 \overset{*}{\Rightarrow} w_n.$$

Let us also introduce a shorthand notation for a set of production rules in the form

$$\begin{gathered} A \to w_0 \\ A \to w_1 \\ \vdots \\ A \to w_n, \end{gathered}$$

which we write as

$$A \to w_0 | w_1 | \cdots | w_n.$$

**Definition 15.** Let $G = (V_T, V_N, S, P)$ be a grammar. The set

$$\mathbf{L}(G) = \left\{ w \in {V_T}^* : S \overset{*}{\Rightarrow} w \right\}$$

is the *language* generated by $G$.

Chapter 2

# Knight's Move Grammar

Cerman in [11] describes a method to build a combinatorial pyramid by choosing contraction kernels based on minimum contrast of the edges. This method creates a structure aware pyramid, where higher levels keep information about the critical points (maxima, minima and saddle points) by letting the associated vertices survive contraction. Contraction kernels in [11] are defined by singletons for both the set of surviving vertices and the set of non-surviving edges, thus the approach is sequential and the height of the pyramid is linear with respect to the number of edges in the base graph. Batavia, Gonzalez-Diaz, and Kropatsch in [3] solve these issues by proposing a parallel approach for the application of contraction kernels, which in turn enables the definition of larger contraction kernels made of independent vertices and edges. These contraction kernels are still based on edge contrast, with random sapling of independent edges. In [2], the authors propose an objective function to choose optimal contraction kernels.

These approaches shine in the generic case of color images, which is where most other pyramid schemes have been historically applied. However, in binary and multi-label images, the choice of edges to contract based on contrast is purely arbitrary. In plateau regions, all edges have the same zero-valued contrast, and these have the same priority in any search algorithm [2]. To solve these issues, the authors in [2] propose to adapt the $3 \times 3 - 4/5$ pyramid scheme (see Fig. 1.4c) to irregular pyramids. This scheme resembles the L-shape move of a knight's piece in Chess shown in Fig. 2.1, thus we will call it *knight's move scheme* in the rest of this work.

In Section 2.1, we propose how to adapt the knight's move scheme to an irregular pyramid scheme. In Section 2.2, we prove some of the scheme's properties and present other motivations to its usage. Finally, in Section 2.3

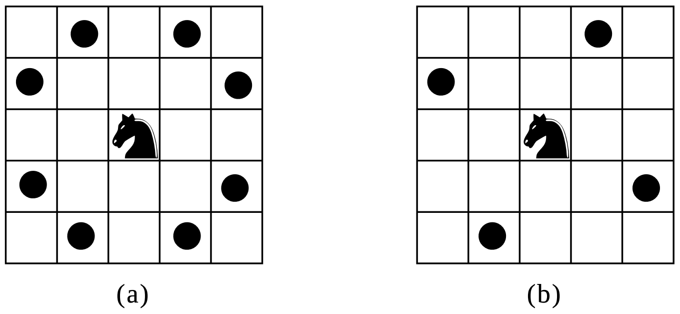


Figure 2.1: Knight's Piece Possible Moves in the Game of Chess. (a) All possible moves. (b) Moves that represent the pyramid scheme.

we propose formal grammars aimed at designing a possible algorithm to traverse the graph and select surviving vertices to generate a knight's move contraction kernel.

## 2.1 Adapting the Scheme

To apply the knight's move scheme to a grid graph $G = (V, E)$, or to the equivalent combinatorial map, we need to generate a contraction kernel $\mathcal{K}$ that resembles such scheme. Let us assume that we can select vertices of the graph with the same pattern of the scheme, we will denote this as the set of surviving vertices $\mathcal{S}$. The set of non-surviving edges will be the set of all edges incident to $v \in \mathcal{S}$ that are not self-loops, that is

$$\mathcal{N} = \{e = (v, u) | v \neq u \wedge I(e) \in \mathcal{S}\},$$

where the operator $I$ denotes the incident vertex of the edge. An example of the application of this contraction kernel to a plateau region can be found in Fig. 2.2.

For the intents and purposes of this work, we are going to exclude all edges that have contrast $c(e)$ greater than zero, therefore

$$\mathcal{N} = \{e = (v, u) | v \neq u \wedge I(e) \in \mathcal{S} \wedge c(e) = 0\}.$$

This is done to preserve topological information about the labels of each plateau region.

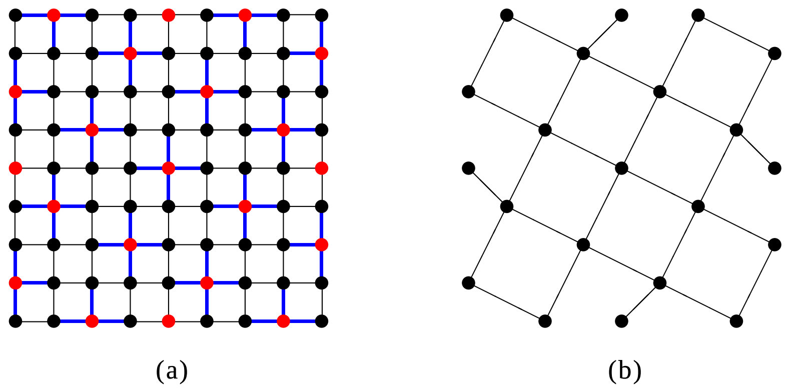


Figure 2.2: Knight's Move Contraction Kernel and Result. (a) Red vertices are the set of surviving vertices $\mathcal{S}$; blue edges are the set of non-surviving edges $\mathcal{N}$. (b) The result of the contraction, which resembles the structure of a grid graph.

## 2.2 Properties of the Scheme

This scheme has been chosen because of a few outstanding properties with respect to other pyramid schemes. One property is that it makes for efficient contraction kernels that can be applied in two steps in parallel. To prove such a property, we will first show that contracting edges around one surviving vertex is an independent operation with respect to other surviving vertices. Then we will show that contraction around one of these vertices can be done in two steps.

**Proposition 1.** *Let $G = (V, E)$ be a grid graph represented with a combinatorial map, and let $\mathcal{K} = (\mathcal{S}, \mathcal{N})$ be a knight's move contraction kernel. $\forall e \in \mathcal{N}$ incident to $v \in \mathcal{S}$, contraction of edge $e$ is dependent only on edges in $\mathcal{N}$ that are incident to $v$.*

*Proof.* $\forall v \in \mathcal{S}$ corresponding to $\sigma^*(d)$, where $d$ is incident to $v$, the edge $e$ corresponding to $\alpha^*(d)$ is dependent to the following set of edges

$$\left\{\alpha^*(\sigma(d)), \alpha^*(\sigma^{-1}(d)), \alpha^*(\sigma(\alpha(d))), \alpha^*(\sigma^{-1}(\alpha(d)))\right\}.$$

Given that $\alpha^*(\sigma(\alpha(d))), \alpha^*(\sigma^{-1}(\alpha(d))) \notin \mathcal{N}$, the only remaining dependencies are in $\sigma^*(d)$, i.e. they are incident to $v$. $\square$

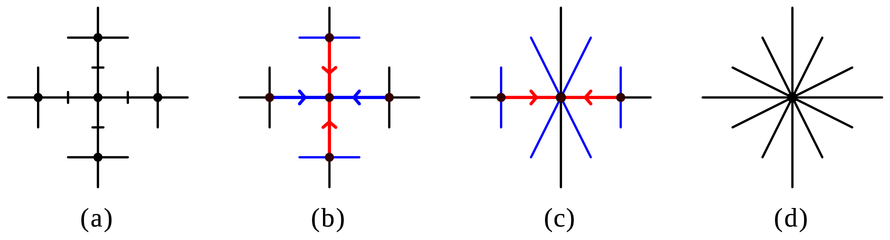


Figure 2.3: Example of Application of a Contraction Kernel on a Grid Graph. The center vertex is the surviving one; in red, selected edges for contraction; in blue, edges dependent on the selected ones. (a) Subgraph of a grid graph. (b) Selection of two independent edges. (c) Selection of remaining two edges. (d) Result.

**Proposition 2.** *Let $G = (V, E)$ be a grid graph encoded by a combinatorial map, and let $\mathcal{K} = (\mathcal{S}, \mathcal{N})$ be a knight's move contraction kernel. Contraction of edges $e \in \mathcal{N}$ incident to an inside vertex $v \in \mathcal{S}$ takes two steps.*

*Proof.* Given the assumptions, Proposition 1 is valid. We can contract two non-consecutive edges in the orbit in one step, e.g. the two vertical ones, because they are independent between each other (see Fig. 2.3b). This operation results in the vertex gaining, as incident edges, its non-survivors' incident edges (see Fig. 2.3c), which makes the remaining two edges in $\mathcal{N}$ independent, and thus they can be contracted in a second step. $\square$

The assumption that the vertex is an *inside* vertex is important. On the boundary, only vertices in the corners can have their incident edges trivially contracted in two steps because they have degree $\deg(v) = 2$. The remaining boundary vertices have degree $\deg(v) = 3$, and, in this case, contraction in two steps is not possible.

**Example 1.** We want to contract every edge around a vertex $v$ of degree $\deg(v) = 3$. Contraction of any one of these edges is dependent on the next edge in the orbit, thus we can choose one of the three as a first step (see Fig. 2.4b). In a second step, it is still not possible to contract the other two in parallel given that contraction of one of the two is dependent on the other. We have to resort in contracting edges sequentially (see Figs. 2.4c to 2.4e), three steps in total given the degree.

Equally important is the grid graph assumption. If any of the edges around the surviving vertex is pendant, the non-surviving vertex at the other endpoint of such an edge will not have any incident vertices by definition.

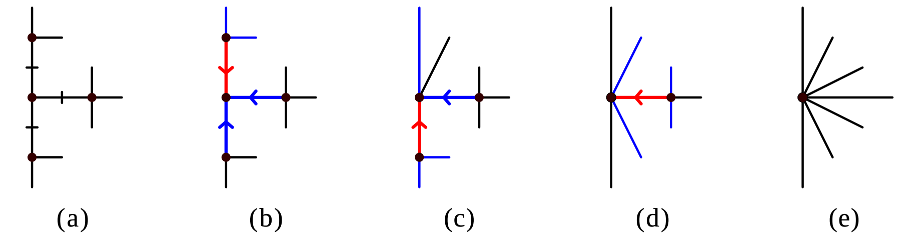


Figure 2.4: Example of Application of a Contraction Kernel on the Boundary. The center vertex is the surviving one; in red, selected edges for contraction; in blue, edges dependent on the selected ones. (a) Generic boundary configuration. (b) First selected edge. (c) Second selected edge. (d) Third selected edge. (e) Result.

Consequently, the surviving vertex will not gain any incident edge after contraction, making the second step of the proof of Proposition 2 impossible. All vertices $v$ of degree $\deg(v) = 4$ in a grid graph will not have any pendant edges incident to it, therefore the importance of the assumption.

Another important property is that this scheme produces another slightly rotated grid graph, on which a new knight's move kernel can be defined (see Fig. 2.2). Also, the fact that we are choosing surviving vertices using a precise scheme makes vertex selection non-arbitrary on plateau regions.

*Note.* Every assumption of a grid graph also applies when we are operating in a subgraph $G'$ of $G$, where $G'$ is a grid graph, but $G$ is not. This is the case of the rotated grid, which is not a grid graph per se.

## 2.3 Vertex Selection on Combinatorial Maps

Until now, we have assumed that we could select vertices in the pattern of a knight's move. Now, we want to describe how this could be achieved on a grid graph and on the consequent graphs that are produced after contraction. On the first level, we might be able to select vertices using geometric information about the input image. This would not be the case for higher levels of the pyramid though, because the grid graph assumption might not always hold and geometric information will not be usable. Thus, we have to resort to navigating the graph and selecting vertices by moving on it like a knight's piece would on a real chessboard.

To navigate the combinatorial map we can use consecutive applications of the involution $\alpha$ and the permutation $\sigma$ on a starting dart $d_s$. Let $\Sigma = \{\alpha, \sigma\}$

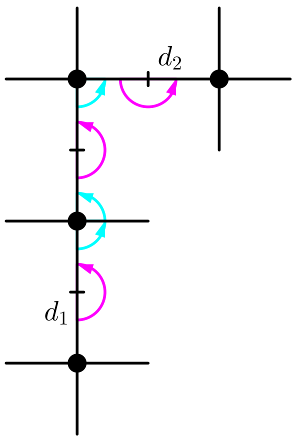


Figure 2.5: Knight's Move on a Combinatorial Map. The starting dart is $d_1$. In magenta are represented $\alpha$ applications, whilst in cyan $\sigma$ applications.

be an alphabet where the symbols refer to their respective function, and let $d_s$ be the starting dart. We can see a word $w \in \Sigma^*$ as the composition of the functions associated to its symbols, applied to $d_s$.

**Example 2.** If $d_1$ is the starting dart and $w = \alpha\sigma$, then the ending dart

$$d_2 = \alpha(\sigma(d_1)).$$

With this approach, we can define a word for the knight's move on the graph. With respect to Fig. 2.5, let $d_1$ be the starting dart and $d_2$ be the wanted ending dart for the knight's move. It holds that

$$d_2 = \alpha\left(\sigma\left(\alpha\left(\sigma^2\left(\alpha\left(d_1\right)\right)\right)\right)\right).$$

Thus, the word that represents the chosen knight's move is

$$\alpha\sigma\alpha\sigma^2\alpha.$$

If we can compose words with the alphabet $\Sigma$, it is possible to define a grammar $\Gamma$ such that each word $w$ in the language $\mathbf{L}(\Gamma)$, applied to a starting dart $d_s$, results in and ending dart $d_e$ such that $I\left(d_e\right)$ is a vertex that respects the pattern of the knight's move.

One of the possible grammars is hereby defined as

$$\Gamma = \left(\{\alpha, \sigma\}, \{M, S\}, S, P\right)$$
$$P = \begin{Bmatrix} M \rightarrow \alpha\sigma\alpha\sigma^2\alpha, \\ S \rightarrow SM\sigma^* \mid \varepsilon \end{Bmatrix}.$$

The $M$ production rule is used as a shorthand to the substitution of a knight's move. The $S$ production rule (1) applies a rotation around the (sigma)-*orbit* of the starting dart, (2) applies the knight's move and (3) one can either repeat the cycle by substituting $S$ with the same production rule, or stop by substituting $S$ with $\varepsilon$.

The language $\mathbf{L}(\Gamma)$ is general enough that each word $w$ represents a possible walk on the graph that ends in a vertex that respects the pattern. This grammar also produces different words $w \in \mathbf{L}(\Gamma)$ that result in visiting the same vertices in the pattern with different walks.

For sequential computing, one might be interested in producing a single word that reaches each vertex of the pattern on a given grid graph. On the other hand, for parallel computing we are interested in a grammar that lets us produce multiple words such that (1) $\mathbf{L}(\Gamma)$ covers all the vertices of the pattern on the graph, and (2) $\exists!\, w \in \mathbf{L}(\Gamma)$ that results in the ending dart $d_e$ after application on the starting dart $d_s$. We propose such grammar here

$$\Gamma = \big(\{\alpha,\sigma\}, \{M, S, U', U''\}, S, P\big)$$
$$P = \left\{\begin{aligned} &M \rightarrow \alpha\sigma\alpha\sigma^2\alpha, \\ &S \rightarrow \underbrace{U'M\sigma^*}_{\text{subst. 1}} \mid \varepsilon, \\ &U' \rightarrow \underbrace{U'M\sigma^{-1}}_{\text{subst. 2}} \mid \underbrace{U''M}_{\text{subst. 3}} \mid \varepsilon, \\ &U'' \rightarrow \underbrace{U''M\sigma^{-1}}_{\text{subst. 4}} \mid \varepsilon \end{aligned}\right\}. \tag{2.1}$$

Production rule $M$ is a shorthand for the knight's move. Production rule $S$ starts the propagation with all the darts in $\sigma^*(d_s)$ and applies the move on each of them, and with the first option of production rule $U'$ four branches are generated that will reach the boundaries of the grid. From these four branches, other secondary branches will be generated by using substitution 3 (production rule $U'$), and with $U''$ we make sure that no more branches are created from these secondary ones. An illustration of the coverage achieved by this grammar can be seen in Fig. 2.6.

We will use this grammar to generate and apply all the words in parallel by using computing approaches like jump flooding, described by Rong and Tan in [16]. Further details are explained in Section 3.3.

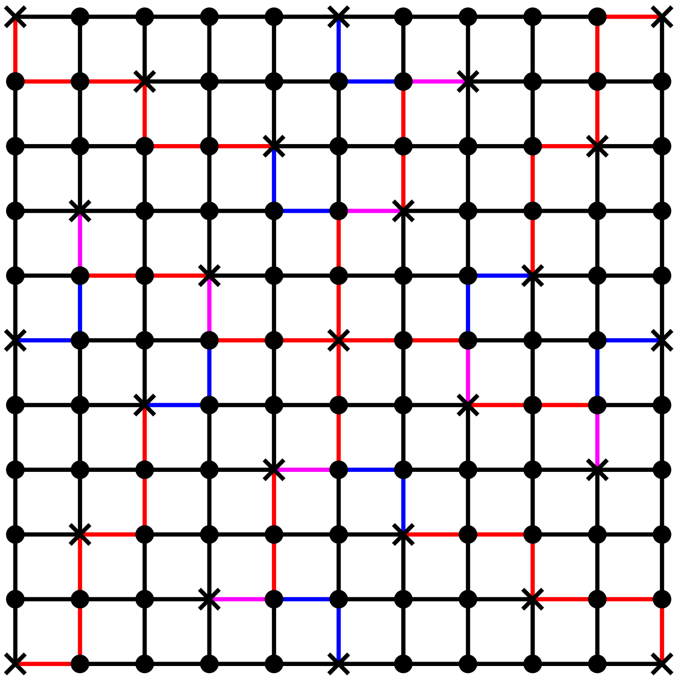


Figure 2.6: Illustration of The Application of The Grammar. With respect to (2.1): the starting dart $d_s$ is one of the darts incident to the vertex in the center; crossed vertices are selected by language; edges in red represent the walks generated by substitutions 1, 2 and 4, where substitution 1 happens only around $d_s$; in blue, the branching moves generated by substitution 3; in magenta, edges covered by both red and blue walks.

# Chapter 3

# Implementation

In this chapter we propose a Connected Component to showcase the properties of the knight's move scheme, and we go over the logic of its functioning.

In Section 3.1 we describe the data structure that holds the combinatorial pyramid. In Section 3.2 we go over the proposed algorithm in a general way, and then in Section 3.3 we delve into the details of some of the choices that have been made regarding the contraction kernel generation and how to deal with some practical problems.

## 3.1 Combinatorial Pyramid Data Structure

In this implementation, the combinatorial pyramid is encoded following the canonical representation described in [18]. On initialization, the grid graph is represented with a combinatorial map where

- darts of an edge are numbered giving precedence to the upper or left dart, on vertical and horizontal edges respectively;
- darts of an edge are numbered sequentially following the edges' order, vertical edges first, then horizontal ones, top to bottom, left to right;
- we refer to edges by the index of their odd dart;
- vertices are numbered top to bottom, left to right;

An example of the numbering system is shown in Fig. 3.1.

We also store and keep updated the inverse permutation $\sigma^{-1}$ for convenience.

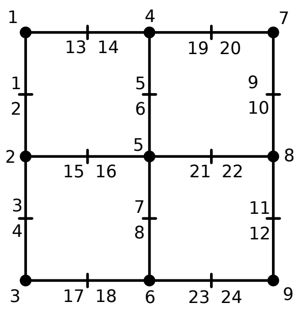


Figure 3.1: Example of Vertex and Dart Numbering.

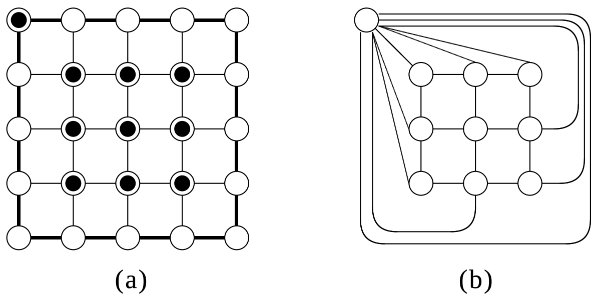


Figure 3.2: Boundary Contraction in a 5 × 5 Image. (a) Contraction Kernel: black vertices denote the set of surviving vertices $\mathcal{S}$ and thick lines denote the set of non-surviving edges $\mathcal{N}$. (b) Result.

## 3.2 Connected Component Labeling Algorithm

Here we propose an algorithm for Connected Component Labeling based on the Knight's Move approach on combinatorial maps. The pseudocode of this algorithm is presented in Algorithm 1. This gets inspiration from the Topology-preserving Irregular Image Pyramid (TIIP) algorithm presented in [3], which is also used as fallback when no efficient contraction kernel is found.

As the authors describe in [3], boundary should be contracted first. In practice, all boundary edges with weight zero are selected for contraction (see Fig. 3.2). This solves an issue in detecting empty self-loops in the graph without resorting to the dual. With respect to Fig. 3.3, if subgraph $A$ was empty, it is easily proven that using Definition 6 makes the two configurations indistinguishable. Still, the only case in which such a configuration is expected is when the boundary of the image is a connected component, which reduces to a non-empty self-loop. Definition 6 would categorize this self-loop as

**Algorithm 1** Connected Component Labeling

```
procedure REDUCE($P$)
    $C \leftarrow$ Combinatorial map generated from $P$
    $C \leftarrow$ Contract all boundary edges $e_b$ of $C$ such that $c(e_b) = 0$
    $s \leftarrow$ Central vertex of $C$
    while edges in $C$ can be contracted do
        $\mathcal{S} \leftarrow$ KNIGHT'S MOVE GRAMMAR(C, s)
        $\mathcal{N} \leftarrow \{e \mid e \in I(\mathcal{S}) \wedge c(e) = 0\}$
        Remove edges from $\mathcal{N}$ with both endpoints $\in \mathcal{S}$    ▷ incl. self-loops
        if $\mathcal{N}$ is non-optimal then
            if no new seed is available then
                break
            else
                $s \leftarrow$ a new seed vertex
                continue
            end if
        end if
        $\mathcal{K} \leftarrow (\mathcal{S}, \mathcal{N})$
        $C \leftarrow$ CONTRACT($C, \mathcal{K}$)
        $C \leftarrow$ REMOVE($C$) ▷ remove empty self-loops and redundant edges
    end while
    if edges in $C$ can be contracted then
        $C \leftarrow$ TIIP(C)
    end if
    if boundary edges are empty self-loops then
        remove these boundary edges from $C$
    end if
    return $C$
end procedure
```

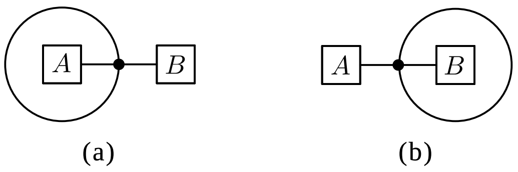


Figure 3.3: Self-loop Configurations Indistinguishable without Dual Graph Representation. $A$ and $B$ are generic subgraphs, which can be empty.

empty. By contracting the boundary first and by ignoring boundary edges in the function Remove, this self-loop will not be wrongly removed. In case the image is a single connected component, in the end this self-loop would become empty and should be removed.

## 3.3 Contraction Kernel Generation

In Algorithm 2 we explain how to implement the graph traversal grammar described in Section 2.3. With this algorithm, we produce a set of surviving vertices $\mathcal{S}$ on the combinatorial map. $I(\mathcal{S})$ denotes the set of incident edges of all vertices in $\mathcal{S}$. The contraction kernel will be defined as seen in Section 2.1, but we cannot assume that the whole graph will be a grid graph, and we also cannot check that we are operating in a subgraph where this assumption holds. Some edges will have to be filtered in order to deal with non-typical cases.

We filter edges that have both endpoints in $\mathcal{S}$, because we want all vertices in $\mathcal{S}$ to survive. This procedure also happens to filter self-loops, where both endpoints are the same vertex, which we cannot contract [8]. These will be removed by the Remove function if they are *empty* self-loops. Another approach would have been to contract such edges all the same, which means that some vertices in $\mathcal{S}$ might not survive. In this case, dealing with dependencies would have become more complex, and this goes out of the scope of our work.

### 3.3.1 Dealing with Boundary

After contraction of the boundary, a few high-degree boundary vertices will remain, one for each connected component of the boundary. We want these vertices to survive because they retain information of the boundary, which is needed to distinguish the two self-loops of Fig. 3.3. This is a sane choice because it does not lose topological information.

**Algorithm 2** Knight's Move Grammar with Jump Flooding [16]

```
function KNIGHT'S MOVE GRAMMAR($C, s$)
    $V_e \leftarrow \{s\}$                                   ▷ the set of reached vertices
    $D_s \leftarrow \{d \mid I(d) = s\}$                      ▷ the set of starting darts
    $Q_1 \leftarrow D_s$                                      ▷ prod. rule $S$
    $Q_2, Q_3 \leftarrow \emptyset$
    while $Q_1 \neq \emptyset \vee Q_2 \neq \emptyset \vee Q_3 \neq \emptyset$ do
        for $i = 1 \dots 3$ do
            $D_i \leftarrow$ KNIGHT'S MOVE($C, Q_i$)
            $D_i \leftarrow D_i \setminus \{d \mid d \in D_i : d \text{ satisfies the stopping criterion}\}$
        end for
        $Q_1 \leftarrow \sigma_C^{-1}(D_1)$                   ▷ 1st substitution of prod. rule $U'$
        $Q_2 \leftarrow D_1$                                  ▷ 2nd substitution of prod. rule $U'$
        $Q_3 \leftarrow \sigma_C^{-1}(D_2 \cup D_3)$          ▷ prod. rule $U''$
        $V_e \leftarrow V_e \cup I\left(\bigcup_{i=1}^{3} D_i\right)$
    end while
    return $V_e$
end function
```

To let these vertices survive, we need to contract edges that are incident to these separately. When such an edge is selected for contraction, it is so because the grammar selected a surviving vertex which is the other endpoint, therefore, we would have a double candidate for survival, i.e. the boundary vertex and the selected one. Separating the two operations, means contracting the "inside" edges first, letting the selected vertex survive, and then contract the remaining edge in the other direction, merging the selected vertex into the boundary vertex.

### 3.3.2 Selecting Independent Edges

When applying contraction in parallel, we need to make sure that we are dealing with dependencies between edges in the right way. The permutation $\sigma$ needs to be consistent after every change, and contracting edges that are linked by this function may result in the overwriting of the same variable multiple times, generating inconsistency. This can be dealt with in two ways: (1) by selecting a subset of independent edges to be contracted at once, then dealing with the rest in the same way, or (2) by dealing with each special case separately. In this work we prefer the first approach, because of its simplicity and generality. For our intents and purposes, the assumption of the 2-step contraction is enough, although by mixing both solutions we expect the best outcome in terms of efficiency. Further research would be necessary.

The following method has been adapted from the one used in TIIP algorithm [3]. Given a contraction kernel where $\mathcal{N} = \{\alpha^*(d_1), \ldots, \alpha^*(d_n)\}$, where darts (and edges) are numbered, we can compute a $5 \times n$ matrix of dependent edges, in the form

$$\mathbf{D} = \begin{pmatrix} \alpha^*(d_1) & \cdots & \alpha^*(d_n) \\ \alpha^*(\sigma(d_1)) & \cdots & \alpha^*(\sigma(d_n)) \\ \alpha^*(\sigma(\alpha(d_1))) & \cdots & \alpha^*(\sigma(\alpha(d_n))) \\ \alpha^*(\sigma^{-1}(d_1)) & \cdots & \alpha^*(\sigma^{-1}(d_n)) \\ \alpha^*(\sigma^{-1}(\alpha(d_1))) & \cdots & \alpha^*(\sigma^{-1}(\alpha(d_n))) \end{pmatrix}.$$

A subset of independent non-surviving edges $\mathcal{N}_{\text{ind}} \subseteq \mathcal{N}$ is

$$\mathcal{N}_{\text{ind}} = \left\{ \alpha^*(d_j) \in \mathcal{N} \mid \alpha^*(d_j) = \min_i \{\mathbf{D}_{ij}\} \right\} \quad \text{with } i = 1 \ldots 5,\ j = 1 \ldots n,$$

where $\mathbf{D}_{ij}$ denotes the $i$-th row and $j$-th column element of $\mathbf{D}$, and the minimum operation is related to the edge numbering. $\mathcal{N}_{\text{ind}}$ is thus composed of all edges in $\mathcal{N}$ which have the lowest index between their group of dependencies, namely the column of $\mathbf{D}$ in which they appear as first element. If two edges were to be dependent, they would appear in each other's column, but only one of the two would be the minimum of its column, and thus selected.

Edges in $\mathcal{N}_{\text{ind}}$ will be contracted in parallel, and then we apply the same procedure on the edges that have not been selected. To make sure that the procedure converges, namely that all edges in $\mathcal{N}$ get selected at some point, we substitute $\infty$ in $\mathbf{D}$ in place of the index of edges that are not in $\mathcal{N}$ or that have been selected in previous iterations.

In a general application, even when the assumptions of Proposition 1 do not hold, this method will produce sets of independent edges to contract in parallel. Under the assumptions of Proposition 2, by using the numbering described in Section 3.1, one can easily see that this method produces a two-step contraction that reflects the one described in Proposition 2. Although, different numberings of edges can affect the optimality, determining cases that are between the worst and the best. In the case in which these assumptions do not hold, at least correctness is assured.

# Chapter 4

# Results

In this chapter we are examining the properties of the knight's move scheme. These are shown by the application of the connected component labeling algorithm on plateau region images, which we describe in Section 4.1. In Section 4.2 we also examine some issues of the proposed algorithm on more generic inputs, namely with different objects in the same image. Finally, in Section 4.3 we bring up some points to solve such issues and improve the algorithm, possibly to a more generic approach for connected component labeling.

## 4.1 Execution on Plateau Region

In Fig. 4.2 (Pages 38 to 41) we show an example of the execution of the algorithm on an image with a single black component. One can clearly see the rotating grid that we theorized in Section 2.2. Apart from few complex subgraphs that remain on the corners of the image, where the graph traversal algorithm does not reach, the adaptation of the pyramid scheme seems to hold to the expectations. Further figures have not been added because they would represent the application of the TIIP [3], which is not relevant to this work.

Tables 4.1 to 4.5 (Pages 42 to 44) show that on the first levels where the knight's move is applied, the theorized 5:1 reduction seems to happen, namely where $\approx 80\%$ appears in the removed vertices and edges column. In the following levels, this does not hold anymore due to the more complex subgraphs near the corners, which make up for the higher number of remaining edges and vertices in the graph. Thus, in percentage, with respect to the total number of remaining edges and vertices, the knight's move seems not to be effective; but, considering the figures of the result, we are still seeing the expected simplification of the central grid graph. The number of knight's moves also increases with the size of the plateau, as expected, apart from the $50 \times 50$ and $100 \times 100$ examples, where the number seems to be the same. This might be due to the heuristic that we implemented to check whether we should continue applying the knight's move or not.

The complexities in proximity of the corners appear due to the grid graph assumption and its consequences, that we described in Section 2.2. In this case, the subgraph which retains the grid graph property shrinks at each level due to the rotation of the grid itself, leaving residues after each application of the knight's move.

*Note.* The number of edges contracted during boundary contraction, shown in Tables 4.1 to 4.5 in columns "Remaining Edges" and "Removed Edges", takes into account also the number of multiple edges that are removed afterwards. In an $m \times n$ grid graph there are $2(m-1) + 2(n-1)$ boundary edges, and the number of extra edges with respect to this quantity is the number of these multiple edges. These tend to appear in the corners due to the fact that in a grid graph there are two boundary vertices adjacent to the inner vertex in the corners.

## 4.2 Problems with Objects

The algorithm that we proposed has a few issues when dealing with multiple blobs inside an image. On the first level, the outcome is always the one we expect. Examples of this are in Figs. 4.3a, 4.4a and 4.5a (Pages 45 to 48), where we can notice some complex subgraphs on the boundaries between objects. These complexities are expected, given that we are removing edges from the contraction kernel if these have contrast $c(e) \neq 0$, which is the case for edges on the boundary. In this case, the grid graph assumption holds only inside the blobs.

Given that we traverse the graph by starting from a single seed, and given that from there we try to reach the boundary of the image, the graph traversal algorithm will behave in unexpected ways when "traversing" these complexities on object boundaries. In these subgraphs, the grid graph assumption does not hold, and the application of $\alpha$ and $\sigma$ operations in the same order as with grid graphs will not reach a vertex that respects the knight's move pattern.

An example of this phenomenon can be seen in Figs. 4.3b to 4.3d, where the pattern gets rotated outside the white square and cannot reach some areas of the graph. Also, surviving vertices are selected in non-optimal ways even *inside* the white square, probably because of some branches of the graph traversal "turning back" inside. The latter phenomenon is clearer in Figs. 4.4 and 4.5, where vertices in the center are all selected for survival, deeming the contraction kernel unusable.

## 4.3 Possible Solutions

Could we try to detect plateau regions and use the knight's move just inside those? This does not seem possible due to the fact that, in some way, we would need to pre-compute the connected components or at least part of such information, which would make the irregular image pyramid scheme fruitless. Still, the limit of the single seed is clear. Could we expand from multiple seeds? Without prior information on the shape and number of these plateaus, we cannot place a seed for each one of them. We would need to resort to random seeds. How many? In the best case, the algorithm would detect a number of connected components equal to the number of seeds, less in a worse case. A high number of seeds may solve this issue. But, we cannot make sure that each of these falls inside one plateau. And, if this happens, how do we merge two seeds inside the same plateau? Then, what about optimality? These questions may have seemingly easy answers, but in this work we could not explore these solutions with the needed attention.

Another approach, seemingly more robust and promising, is changing the graph traversal algorithm. This new algorithm would start from the boundary and expand to the inside. This search needs to both select surviving vertices *and* find new seeds dynamically. A new seed candidate could be generated each time an edge with weight greater than 0 is traversed. More research should be done to unveil how these components have to behave to reach an optimal and correct result.

### 4.3.1 Geometrical Vertex Selection

In this work geometrical information was discarded at first because we wanted to take advantage of combinatorial maps' properties. Instead, let us pave the way to approaches that may take advantage of such information.

Let $G$ be an $m \times n$ grid graph where the vertices are labeled by their coordinates in terms of rows and columns, starting from the bottom-left corner with coordinates $(0, 0)$; let $L$ denote the level of the pyramid, where $L = 0$ is the base level. The set of surviving vertices of level $L = 1 \dots H$, $\mathcal{S}_L$,

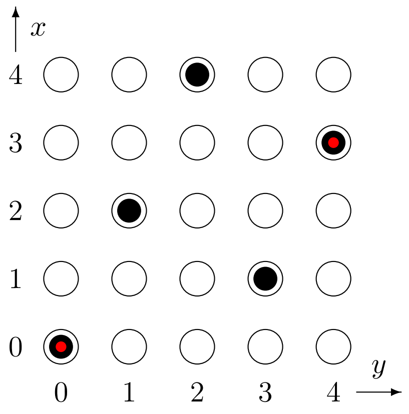


Figure 4.1: Geometrical Vertex Selection. In black, level 1 surviving vertices. In red, level 2 surviving vertices.

is defined by the vertices with coordinates

$$\left(x, \underbrace{\left(5^L - \mathbf{s}_L\right) x \bmod 5^L}_{\text{offset of the row}} + 5^L k\right) \qquad \text{with } x = 0 \ldots m-1,\ k = 0 \ldots \left\lceil \frac{n}{5^L} \right\rceil - 1$$

where $\mathbf{s}_L$ is an offset that depends on the level, the values of which have been deduced from the application of the criterion itself as to be

$$\mathbf{s} = \begin{bmatrix} 2 & 7 & 57 & 182 & 2057 & 14557 & \cdots \end{bmatrix}.$$

See Fig. 4.1 for an illustration. Further research is needed to find out whether an algorithm exists to generate $\mathbf{s}$ for each level. Also, this formula could be generalized to generate a pattern with seed different from $(0, 0)$.

In principle, we may choose every incident edge to the surviving vertices as the set of non surviving edges $\mathcal{N}$. To preserve topology and to solve the issues on the boundary of the image, we propose a few exceptions in the selection of the contraction kernel. We shall not contract any edge that connects an inner vertex to a boundary vertex or two regions with different labels. This may cause some isolated vertices to be left along the boundary (or border). It would be sufficient to integrate the remaining, isolated vertices as survivors or non-survivors that are contracted by a few edges along the boundary such that the overall boundary is preserved as much as possible.

There are only a few cases to be considered:

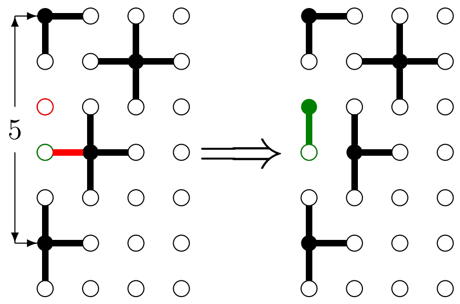
1. two successive survivors along one of the four (straight) boundary segments;

2. two successive survivors around a corner at a distance of 4,3, and 2 to the corner.

Let us discuss a few possible cases around the corners. These exceptions have been theorized for level 1, and should be adapted for higher levels.

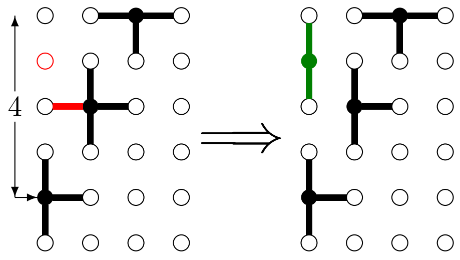
### Adapting sides when the corner survives

Initially, selected survivors along the straight boundary have a distance of 5 edges. Two of the edges are already proposed to be contracted to the survivor on the boundary. The remaining three edges along the boundary are separated by two vertices. The edge between them can be contracted.



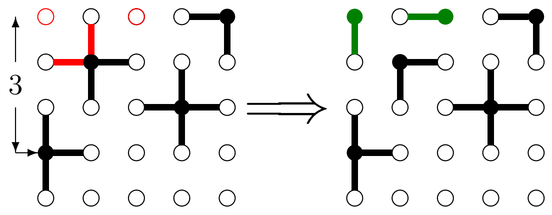
The distance between two survivors along one side is 5. The red elements are removed, the green ones added. The same corrections may be applied to all four sides of the image. The distance of the survivor to the corner is 5.

### Corner at vertical distance 4 to the survivor



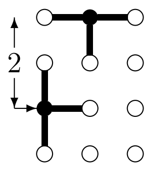
The distance between two survivors around the corner is 6.

### Corner at vertical distance 3 to the survivor



The distance between two survivors around the corner is 7.

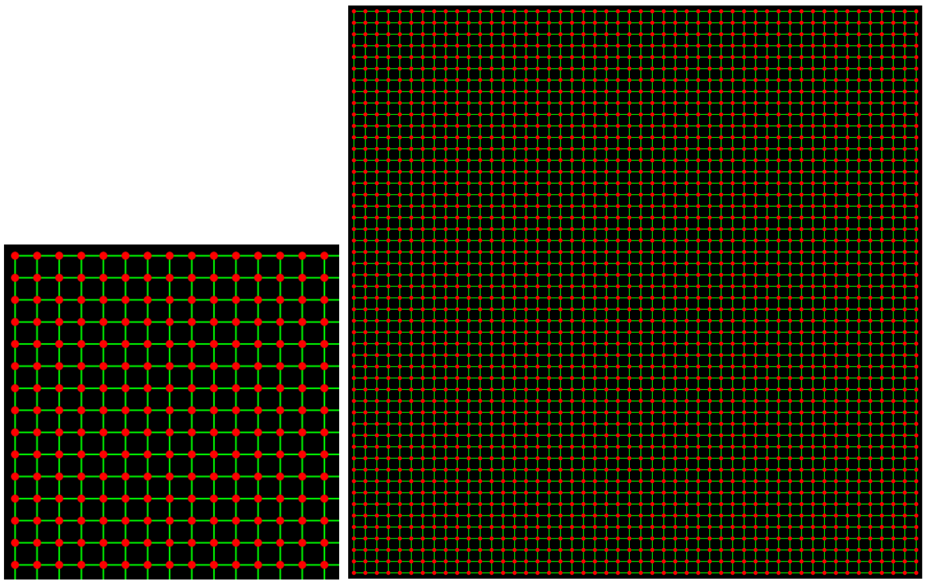
### Corner at vertical distance 2 to the survivor



The distance between two survivors around the corner is 3. There is nothing to change in this configuration, no isolated vertices are created between the two surviving vertices and two of the edges are already proposed for being contracted, the remaining edge survives along the boundary.

### Borders between regions inside the image

Borders between regions with different labels can receive a similar strategy. Edges connecting different region should not be contracted. The difficulty there is that the borders may not be straight in general and need a few more cases to be considered depending on the shapes of the borders. The goal is to sub-sample the border such that the surviving vertices are not moved too far away from the border and that the density in the surrounding of the border corresponds to the remaining density.


(a) base level

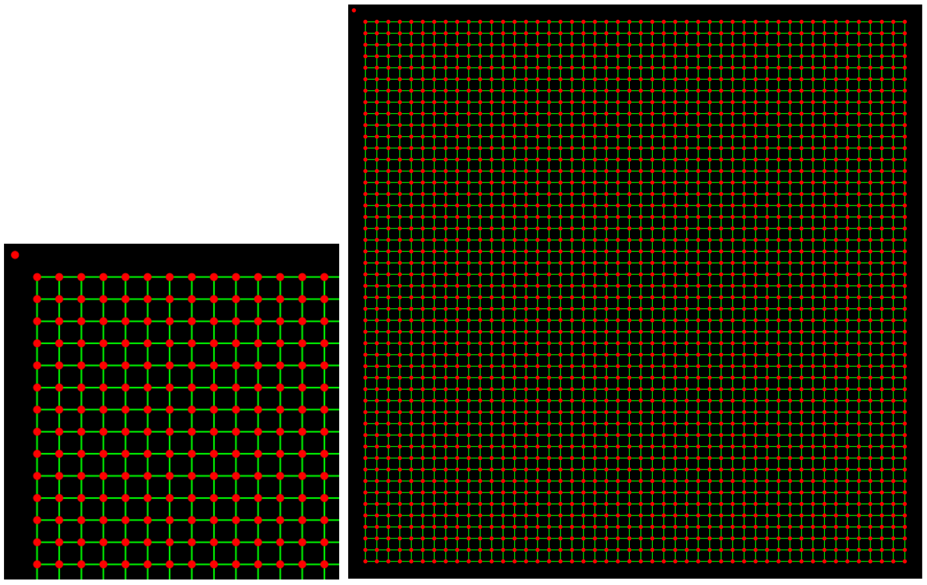
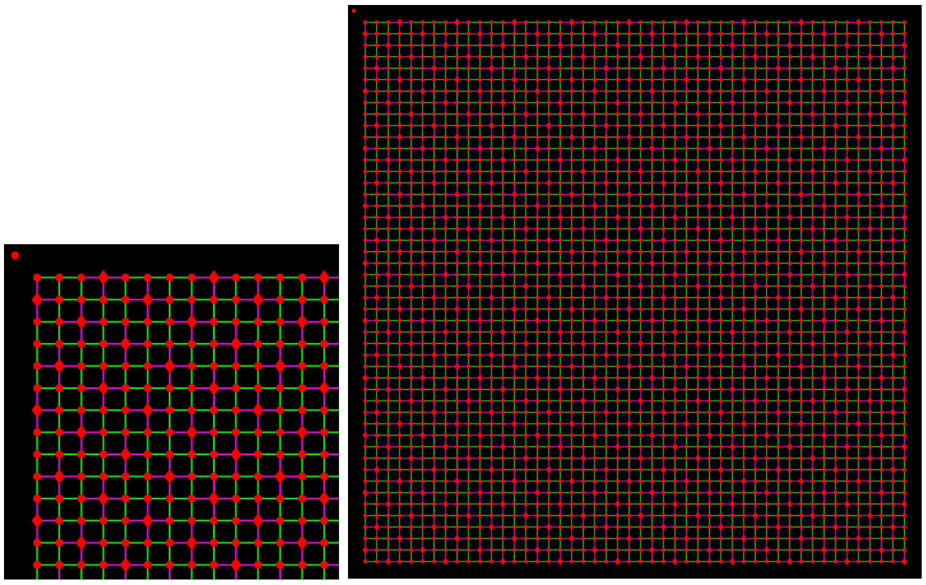
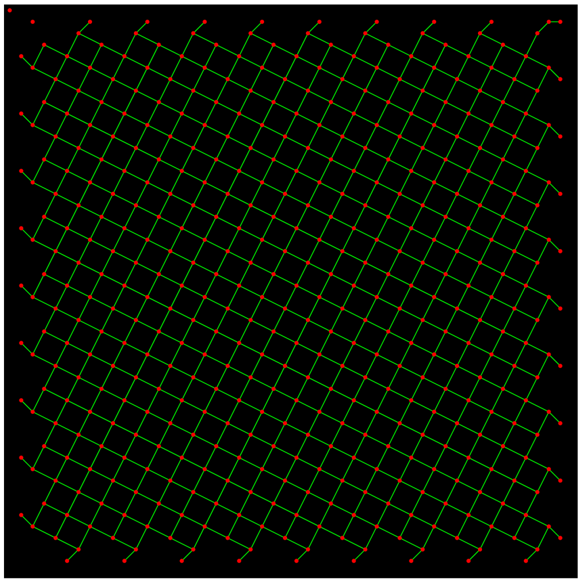

(b) base level with boundary contracted

Figure 4.2: Execution of the Algorithm on a $50 \times 50$ Black Image. Edges that are incident to the single boundary vertex are not shown for better clarity. On the left side, a zoom in the upper-left corner.

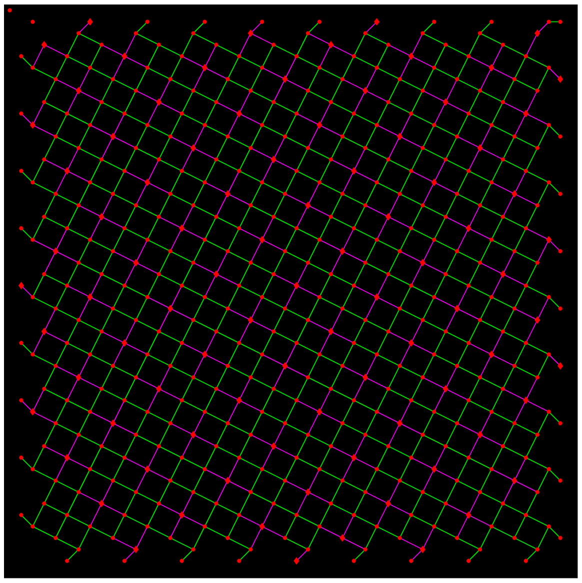


(c) contraction kernel on the base

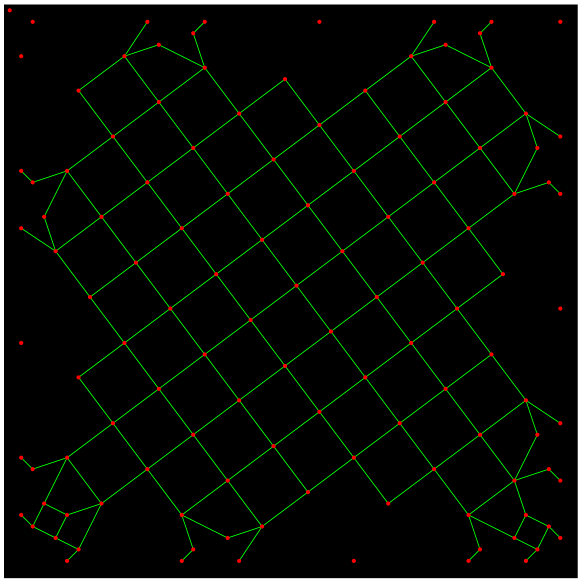


(d) level 1

Figure 4.2: Execution of the Algorithm on a $50 \times 50$ Black Image. Edges that are incident to the single boundary vertex are not shown for better clarity. On the left side, a zoom in the upper-left corner. Lozenge-shaped (⧫) vertices are surviving vertices, and magenta edges are non-surviving edges.

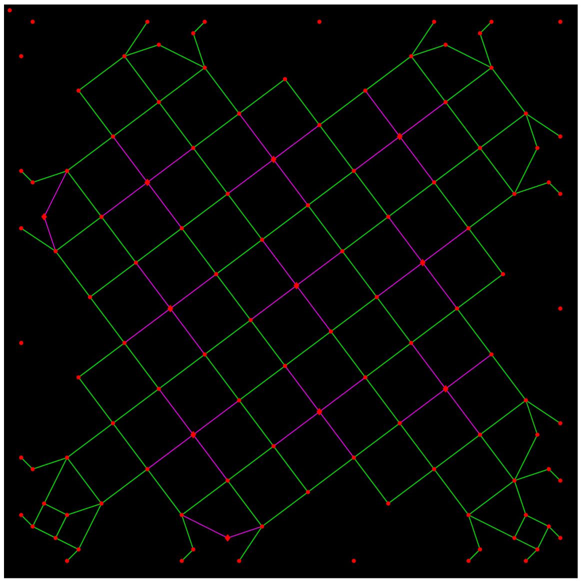


(e) contraction kernel on level 1

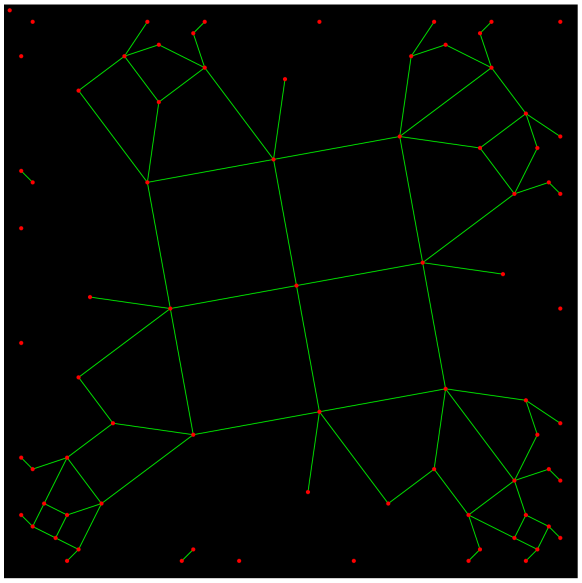


(f) level 2

Figure 4.2: Execution of the Algorithm on a $50 \times 50$ Black Image. Edges that are incident to the single boundary vertex are not shown for better clarity. Lozenge-shaped (⧫) vertices are surviving vertices, and magenta edges are non-surviving edges.



(g) contraction kernel on level 2



(h) level 3

Figure 4.2: Execution of the Algorithm on a $50 \times 50$ Black Image. Edges that are incident to the single boundary vertex are not shown for better clarity. Lozenge-shaped (⧫) vertices are surviving vertices, and magenta edges are non-surviving edges.

Table 4.1: Execution of the Algorithm on a 50 × 50 Black Image.

| | | Remaining | | Removed | | % Removed | |
|---|---|---|---|---|---|---|---|
| Level | Operation | Edges | Vertices | Edges | Vertices | Edges | Vertices |
| 0 | base | 4900 | 2500 | | | | |
| 0 | boundary contr. | 4701 | 2305 | 199 | 195 | 4.06% | 7.80% |
| 1 | knight's move | 941 | 462 | 3761 | 1843 | 79.98% | 79.96% |
| 2 | knight's move | 256 | 120 | 685 | 342 | 72.79% | 74.03% |
| 3 | knight's move | 172 | 78 | 84 | 42 | 32.81% | 35.00% |
| 4 | knight's move | 116 | 51 | 56 | 27 | 32.56% | 34.62% |
| 5 | TIIP | 66 | 29 | 50 | 22 | 43.10% | 43.14% |
| 6 | TIIP | 60 | 26 | 6 | 3 | 9.09% | 10.34% |
| 7 | TIIP | 40 | 18 | 20 | 8 | 33.33% | 30.77% |
| 8 | TIIP | 34 | 15 | 6 | 3 | 15.00% | 16.67% |
| 9 | TIIP | 16 | 7 | 18 | 8 | 52.94% | 53.33% |
| 10 | TIIP | 11 | 5 | 5 | 2 | 31.25% | 28.57% |
| 11 | TIIP | 5 | 3 | 6 | 2 | 54.55% | 40.00% |
| 12 | TIIP | 1 | 1 | 4 | 2 | 80.00% | 66.67% |
| 13 | empty self-loop | 0 | 1 | 1 | 0 | 100.00% | 0.00% |

Table 4.2: Execution of the Algorithm on a 100 × 100 Black Image.

| | | Remaining | | Removed | | % Removed | |
|---|---|---|---|---|---|---|---|
| Level | Operation | Edges | Vertices | Edges | Vertices | Edges | Vertices |
| 0 | base | 19800 | 10000 | | | | |
| 0 | boundary contr. | 19401 | 9605 | 399 | 395 | 2.02% | 3.95% |
| 1 | knight's move | 3881 | 1922 | 15521 | 7683 | 80.00% | 79.99% |
| 2 | knight's move | 953 | 458 | 2928 | 1464 | 75.44% | 76.17% |
| 3 | knight's move | 386 | 176 | 567 | 282 | 59.50% | 61.57% |
| 4 | knight's move | 232 | 102 | 154 | 74 | 39.90% | 42.05% |
| 5 | TIIP | 130 | 57 | 102 | 45 | 43.97% | 44.12% |
| 6 | TIIP | 83 | 36 | 47 | 21 | 36.15% | 36.84% |
| 7 | TIIP | 47 | 21 | 36 | 15 | 43.37% | 41.67% |
| 8 | TIIP | 24 | 10 | 23 | 11 | 48.94% | 52.38% |
| 9 | TIIP | 12 | 6 | 12 | 4 | 50.00% | 40.00% |
| 10 | TIIP | 2 | 2 | 10 | 4 | 83.33% | 66.67% |
| 11 | TIIP | 1 | 1 | 1 | 1 | 50.00% | 50.00% |
| 12 | empty self-loop | 0 | 1 | 1 | 0 | 100.00% | 0.00% |

Table 4.3: Execution of the Algorithm on a 150 × 150 Black Image.

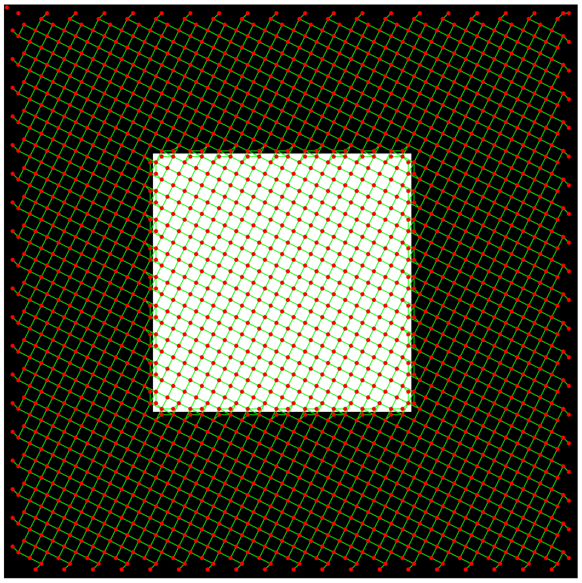
| | | Remaining | | Removed | | % Removed | |
|---|---|---|---|---|---|---|---|
| Level | Operation | Edges | Vertices | Edges | Vertices | Edges | Vertices |
| 0 | base | 44700 | 22500 | | | | |
| 0 | boundary contr. | 44101 | 21905 | 599 | 595 | 1.34% | 2.64% |
| 1 | knight's move | 8821 | 4382 | 35281 | 17523 | 80.00% | 80.00% |
| 2 | knight's move | 2045 | 994 | 6776 | 3388 | 76.82% | 77.32% |
| 3 | knight's move | 772 | 360 | 1273 | 634 | 62.25% | 63.78% |
| 4 | knight's move | 499 | 226 | 273 | 134 | 35.36% | 37.22% |
| 5 | knight's move | 396 | 177 | 103 | 49 | 20.64% | 21.68% |
| 6 | knight's move | 351 | 153 | 45 | 24 | 11.36% | 13.56% |
| 7 | knight's move | 303 | 129 | 48 | 24 | 13.68% | 15.69% |
| 8 | knight's move | 261 | 107 | 42 | 22 | 13.86% | 17.05% |
| 9 | TIIP | 85 | 34 | 176 | 73 | 67.43% | 68.22% |
| 10 | TIIP | 31 | 13 | 54 | 21 | 63.53% | 61.76% |
| 11 | TIIP | 10 | 5 | 21 | 8 | 67.74% | 61.54% |
| 12 | TIIP | 2 | 2 | 8 | 3 | 80.00% | 60.00% |
| 13 | TIIP | 1 | 1 | 1 | 1 | 50.00% | 50.00% |
| 14 | empty self-loop | 0 | 1 | 1 | 0 | 100.00% | 0.00% |

Table 4.4: Execution of the Algorithm on a 200 × 200 Black Image.

| | | Remaining | | Removed | | % Removed | |
|---|---|---|---|---|---|---|---|
| Level | Operation | Edges | Vertices | Edges | Vertices | Edges | Vertices |
| 0 | base | 79600 | 40000 | | | | |
| 0 | boundary contr. | 78801 | 39205 | 799 | 795 | 1.00% | 1.99% |
| 1 | knight's move | 15761 | 7842 | 63041 | 31363 | 80.00% | 80.00% |
| 2 | knight's move | 3565 | 1744 | 12196 | 6098 | 77.38% | 77.76% |
| 3 | knight's move | 1348 | 638 | 2217 | 1106 | 62.19% | 63.42% |
| 4 | knight's move | 850 | 392 | 498 | 246 | 36.94% | 38.56% |
| 5 | knight's move | 767 | 352 | 83 | 40 | 9.76% | 10.20% |
| 6 | knight's move | 687 | 312 | 80 | 40 | 10.43% | 11.36% |
| 7 | knight's move | 633 | 285 | 54 | 27 | 7.86% | 8.65% |
| 8 | knight's move | 552 | 245 | 81 | 40 | 12.80% | 14.04% |
| 9 | knight's move | 491 | 215 | 61 | 30 | 11.05% | 12.24% |
| 10 | knight's move | 432 | 186 | 59 | 29 | 12.02% | 13.49% |
| 11 | knight's move | 385 | 162 | 47 | 24 | 10.88% | 12.90% |
| 12 | TIIP | 141 | 58 | 244 | 104 | 63.38% | 64.20% |
| 13 | TIIP | 60 | 24 | 81 | 34 | 57.45% | 58.62% |
| 14 | TIIP | 7 | 4 | 53 | 20 | 88.33% | 83.33% |
| 15 | TIIP | 1 | 1 | 6 | 3 | 85.71% | 75.00% |
| 16 | empty self-loop | 0 | 1 | 1 | 0 | 100.00% | 0.00% |

Table 4.5: Execution of the Algorithm on a 250 $\times$ 250 Black Image.

| | | Remaining | | Removed | | % Removed | |
|---|---|---|---|---|---|---|---|
| Level | Operation | Edges | Vertices | Edges | Vertices | Edges | Vertices |
| 0 | base | 124500 | 62500 | | | | |
| 0 | boundary cntr. | 123501 | 61505 | 999 | 995 | 0.80% | 1.59% |
| 1 | knight's move | 24701 | 12302 | 98801 | 49203 | 80.00% | 80.00% |
| 2 | knight's move | 5609 | 2756 | 19092 | 9546 | 77.29% | 77.60% |
| 3 | knight's move | 2003 | 956 | 3606 | 1800 | 64.29% | 65.31% |
| 4 | knight's move | 1172 | 544 | 831 | 412 | 41.49% | 43.10% |
| 5 | knight's move | 960 | 435 | 212 | 109 | 18.09% | 20.04% |
| 6 | knight's move | 886 | 398 | 74 | 37 | 7.71% | 8.51% |
| 7 | knight's move | 825 | 366 | 61 | 32 | 6.88% | 8.04% |
| 8 | knight's move | 771 | 339 | 54 | 27 | 6.55% | 7.38% |
| 9 | knight's move | 709 | 306 | 62 | 33 | 8.04% | 9.73% |
| 10 | knight's move | 639 | 269 | 70 | 37 | 9.87% | 12.09% |
| 11 | knight's move | 580 | 241 | 59 | 28 | 9.23% | 10.41% |
| 12 | knight's move | 534 | 218 | 46 | 23 | 7.93% | 9.54% |
| 13 | knight's move | 485 | 196 | 49 | 22 | 9.18% | 10.09% |
| 14 | TIIP | 460 | 181 | 25 | 15 | 5.15% | 7.65% |
| 15 | TIIP | 448 | 174 | 12 | 7 | 2.61% | 3.87% |
| 16 | TIIP | 134 | 52 | 314 | 122 | 70.09% | 70.11% |
| 17 | TIIP | 126 | 49 | 8 | 3 | 5.97% | 5.77% |
| 18 | TIIP | 52 | 22 | 74 | 27 | 58.73% | 55.10% |
| 19 | TIIP | 30 | 13 | 22 | 9 | 42.31% | 40.91% |
| 20 | TIIP | 23 | 10 | 7 | 3 | 23.33% | 23.08% |
| 21 | TIIP | 17 | 7 | 6 | 3 | 26.09% | 30.00% |
| 22 | TIIP | 2 | 2 | 15 | 5 | 88.24% | 71.43% |
| 23 | TIIP | 1 | 1 | 1 | 1 | 50.00% | 50.00% |
| 24 | empty self-loop | 0 | 1 | 1 | 0 | 100.00% | 0.00% |


(a) level 1

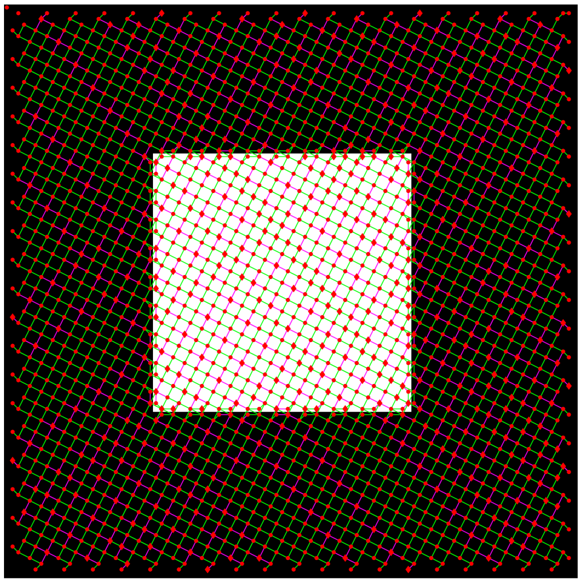
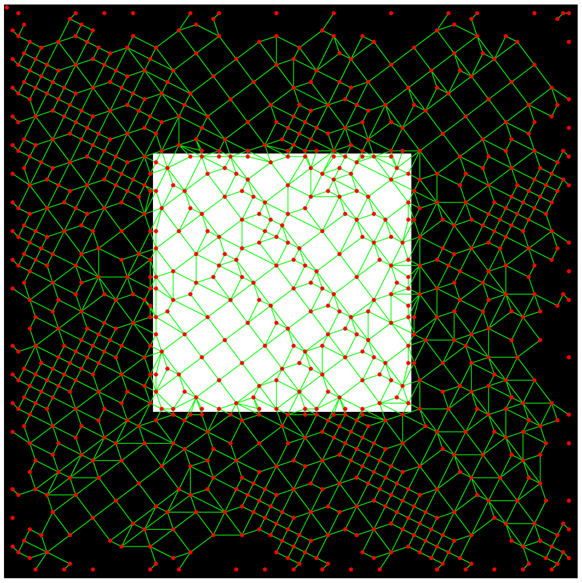
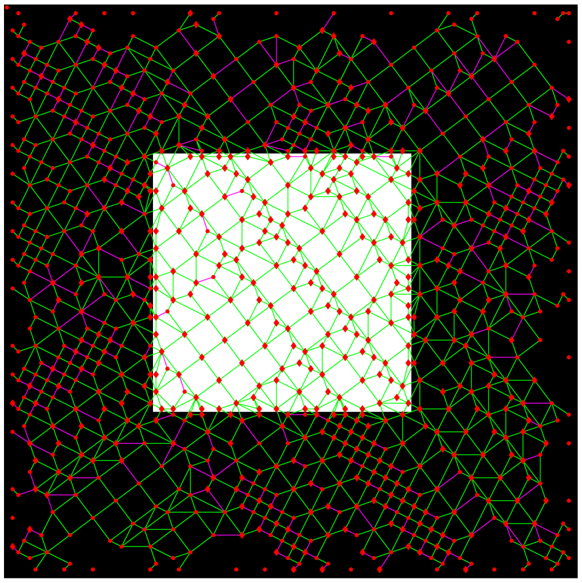

(b) contraction kernel on lv. 1

Figure 4.3: Knight's Move Rotation Around the Boundary of Objects. Edges that are incident to a border vertex are not shown for better clarity. Lozenge-shaped (⧫) vertices are surviving vertices, and magenta edges are non-surviving edges.

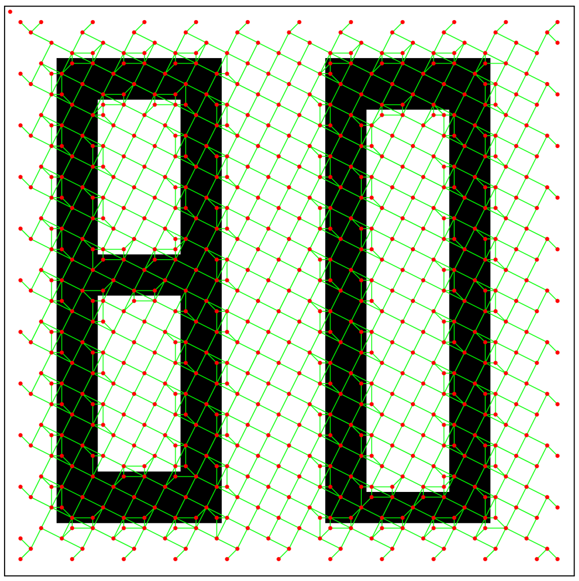


(c) level 2

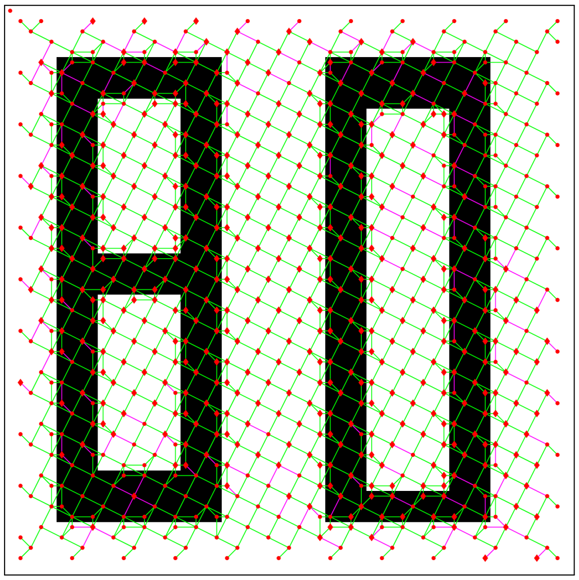


(d) contraction kernel on lv. 2

Figure 4.3: Knight's Move Rotation Around the Boundary of Objects. Edges that are incident to a border vertex are not shown for better clarity. Lozenge-shaped (♦) vertices are surviving vertices, and magenta edges are non-surviving edges.

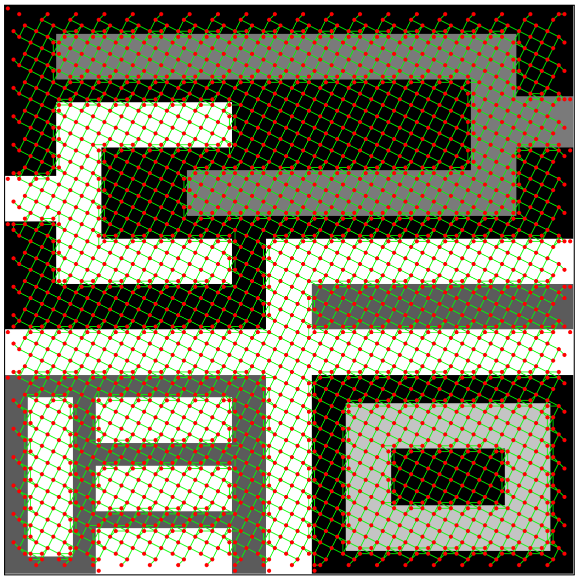


(a) level 1



(b) contraction kernel on lv. 1

Figure 4.4: Knight's Move Excessive Vertex Selection. Edges that are incident to a border vertex are not shown for better clarity. Lozenge-shaped (⧫) vertices are surviving vertices, and magenta edges are non-surviving edges.


(a) level 1

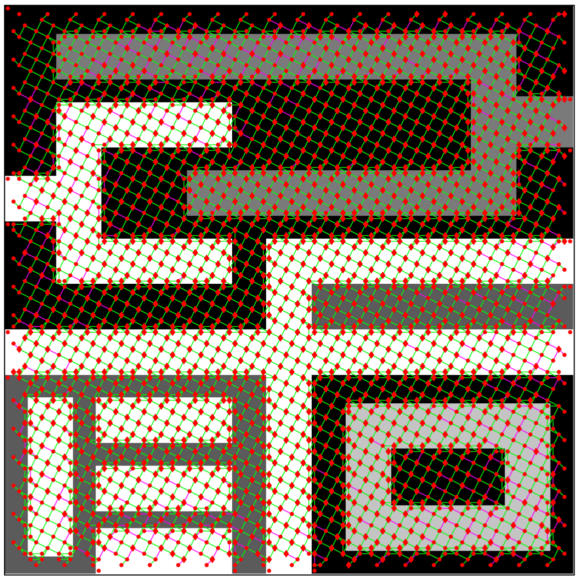

(b) contraction kernel on lv. 1

Figure 4.5: Knight's Move Excessive Vertex Selection. Edges that are incident to a border vertex are not shown for better clarity. Lozenge-shaped (⧫) vertices are surviving vertices, and magenta edges are non-surviving edges.

# Conclusions

In this work we proposed a novel method for contraction kernel generation in binary and multi-label images. We adapted a scheme for regular image pyramids to irregular image pyramids, based on combinatorial map representation of graphs.

The knight's move scheme generates contraction kernels that can be applied in two steps, thanks to the peculiar pattern of vertices selection that guarantees independent sets of edges.

The defined rules with which we choose surviving vertices makes the outcome predictable, and, thanks to the rotating grid, the pattern can be applied at each level, until the assumption of the grid graph does not hold anymore. This also means that vertices are not chosen arbitrarily inside the plateau regions.

We proposed an algorithm for Connected Component Labeling that takes advantage of the knight's move scheme to showcase these properties. Through execution on benchmark images, we showed that the theorized properties actually hold on a practical example, although with some caveats. Our algorithm is not optimal on images with different objects, and we propose a new algorithm design to solve these problems. Still, we showed that, on plateau regions, the pattern achieves properties similar to its regular pyramid's counterpart, namely its reduction factor and the resulting pyramid height.

## Further Developments

We propose the design and the implementation of the Connected Component Algorithm proposed in Section 4.3, which should dynamically find separate plateau regions where to apply the pattern, or, alternatively, we propose to further investigate the usage of geometrical information.

The knight's move scheme is defined for regular image pyramids, and we adapted it to 2-dimensional combinatorial maps. 3-dimensional and $n$-dimensional combinatorial map representations could benefit of a similar scheme, so that we could define irregular pyramids also for 3-dimensional images and $n$-dimensional data structures. We could also shift the attention to $n$-dimensional generalized maps ($n$-Gmaps) which are generalizations of the combinatorial map data structure, and on which similarly defined reduction schemes are applied.

It is possible to apply the combinatorial pyramid for tasks of pattern recognition for segmented images by means of comparisons between graphs. We propose to investigate such scenarios, like object tracking and scene detection, where topological information and relationships between segmented objects might improve prediction results of current systems.

Finally, we propose to investigate lossy and lossless compression of combinatorial pyramids by means of level approximation. For lossless compression, it is possible to compare the expected grid graph of the knight's move to the actual outcome of the algorithm. We expect that storing the difference between the two is more efficient. For lossy compression, some level's information might be outright deleted from the data structure, and it could be accessed through some level prediction algorithm. This is only possible thanks to the predictability of the scheme.

# Bibliography

[1] Edward H. Adelson, Charles H. Anderson, James R. Bergen, Peter J. Burt, and Joan M. Ogden. “Pyramid methods in image processing”. In: *RCA Engineer* 29.6 (Nov./Dec. 1984), pp. 33–41.

[2] Darshan Batavia, Rocio Gonzalez-Diaz, and Walter G. Kropatsch. “A Step Towards Learning Contraction Kernels for Irregular Pyramids”. In: *Proceedings of the 11th International Conference on Pattern Recognition Applications and Methods - ICPRAM*. Vol. 1. INSTICC. SciTePress, Feb. 2022, pp. 60–70. ISBN: 978-989-758-549-4. DOI: 10.5220/0010840900003122.

[3] Darshan Batavia, Rocio Gonzalez-Diaz, and Walter G. Kropatsch. “Image = structure + few colors”. In: *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*. Ed. by Antonio Robles-Kelly. Springer International, Jan. 2021, pp. 365–375.

[4] Michel Bister, Jan Cornelis, and Azriel Rosenfeld. “A critical view of pyramid segmentation algorithms”. In: *Pattern Recognition Letters* 11.9 (1990), pp. 605–617.

[5] Luca Boccia. “Novel strategies for contraction kernel generation in combinatorial maps”. MA thesis. Università degli Studi di Salerno, Feb. 2022.

[6] Luc Brun and Walter G. Kropatsch. “Dual contraction of combinatorial maps”. In: *2nd IAPR-TC-15 Workshop on Graph-based Representation*. Ed. by Walter G. Kropatsch and Jean Michel Jolion. Österreichische Computer Gesellschaft. OCG-Schriftenreihe, 1999, pp. 145–154.

[7] Luc Brun and Walter G. Kropatsch. “Introduction to combinatorial pyramids”. In: *Digital and image geometry*. Ed. by Gilles Bertrand, Atsushi Imiya, and Reinhard Klette. Vol. 2243. Heidelberg: Springer Berlin, 2001, pp. 108–128.

[8] Luc Brun and Walter G. Kropatsch. “Irregular Pyramids with Combinatorial Maps”. In: *Advances in Pattern Recognition, Joint IAPR International Workshops on SSPR’2000 and SPR’2000.* Ed. by Francesc J. Ferri, José M. Iñesta, Adnan Amin, and Pavel Pudil. Vol. 1876. Alicante, Spain: Springer Berlin Heidelberg, Aug. 2000, pp. 256–265. ISBN: 978-3-540-44522-7.

[9] Peter J. Burt and Edward H. Adelson. “The Laplacian Pyramid as a Compact Image Code”. In: *Readings in Computer Vision.* Ed. by Martin A. Fischler and Oscar Firschein. San Francisco (CA): Morgan Kaufmann, 1987, pp. 671–679. ISBN: 978-0-08-051581-6. DOI: `10.1016/B978-0-08-051581-6.50065-9`. URL: `https://www.sciencedirect.com/science/article/pii/B9780080515816500659`.

[10] Peter J. Burt, Tsai-Hong Hong, and Azriel Rosenfeld. “Segmentation and estimation of image region properties through cooperative hierarchial computation”. In: *IEEE Transactions on Systems, Man, and Cybernetics* 11.12 (1981), pp. 802–809.

[11] Martin Cerman. *Structurally Correct Image Segmentation using Local Binary Patterns and the Combinatorial Pyramid.* Tech. rep. PRIP-TR-133. PRIP, TU Wien, 2015. URL: `https://www.prip.tuwien.ac.at/pripfiles/trs/tr133.pdf`.

[12] John Robert Jr Edmonds. “A combinatorial representation for oriented polyhedral surfaces”. PhD thesis. University of Maryland, 1960. DOI: `10.13016/daw5-mvla`. URL: `http://hdl.handle.net/1903/24820`.

[13] Walter G. Kropatsch. “From equivalent weighting functions to equivalent contraction kernels”. In: *Czech Pattern Recognition Workshop ’97.* Czech Pattern Recognition Society. Milovy, Feb. 1997, pp. 1–13.

[14] Peter Linz. *An introduction to formal languages and automata.* 6. Jones & Bartlett Learning, 2017, p. 450. ISBN: 9781284077247.

[15] Annick Montanvert, Peter Meer, and Azriel Rosenfeld. “Hierarchical image analysis using irregular tessellations”. In: *European Conference on Computer Vision.* Springer. 1990, pp. 28–32.

[16] Guodong Rong and Tiow-Seng Tan. “Jump Flooding in GPU with Applications to Voronoi Diagram and Distance Transform”. In: *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games.* I3D ’06. Redwood City, California: Association for Computing Machinery, 2006, pp. 109–116. ISBN: 978-1-59593-295-2. DOI: `10.1145/1111411.1111431`.

[17] Azriel Rosenfeld. *Multiresolution image processing and analysis*. Vol. 12. Springer Series in Information Sciences. Springer Science & Business Media, 2013. ISBN: 978-3-642-51592-7. DOI: `10.1007/978-3-642-51590-3`.

[18] Fuensanta Torres and Walter G. Kropatsch. "Canonical Encoding of the Combinatorial Pyramid". In: *Proceedings of the 19th Computer Vision Winter Workshop 2014*. Ed. by Zuzana Kúleková and Jan Heller. Krtiny, CZ, Feb. 2014, pp. 118–125. ISBN: 978-80-260-5641-6.