UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Solving problems by taking pictures

Study of general graph matching approaches

Dissertation presented by
Benjamin DE WERGIFOSSE , Alexis PIERRET

for obtaining the Master's degree in
Computer Science and Engineering

Supervisor
Pierre SCHAUS

Reader
Yves DEVILLE, Cyrille DEJEMEPPE

Academic year 2015-2016

Abstract

In a world where new technologies emerge every year, the field of computer vision becomes more and more important. Robots, autonomous cars, video surveillances with face recognition, licence plate recognitions are all examples where computer vision techniques apply to detect and understand real life objects. This understanding aims at extracting useful information and possibly make computations with it. Most of the time it is to ease, accelerate or automate the work of humans in different tasks. Due to the diversity of the application fields, most existing techniques are specific to a given problem.

In this thesis, different approaches from specific to general have been considered. The main part of the study focuses on relatively general approaches making use of *graph matching* techniques. The first step consists in detecting, from the input image, a graph composed of nodes and edges, called the *sGraph*. Then, given a graph description of the problem to identify (called the *mGraph*), the aim is to find the largest subgraph of *sGraph* that matches, according a *matching* measure, a subgraph of *mGraph*. Unfortunately, this problem is *NP-complete* and requires an exponential computation time to solve it completely. It is then important to make decent assumptions and to rely on good heuristics to find a good (but not necessary the best) solution of the problem within a reasonable time.

In this work, different *graph matching* approaches are presented and two of them give satisfactory results on relatively large graphs. The first one is *coordinates dependant* and rely on the *sGraph* nodes coordinates and edges attributes to perform the search. The second approach is *coordinates independent* and only uses the traditional graph structure (nodes and edges) to perform the search and find the best matching. Each of these two techniques has its advantages and disadvantages according the topology of researched problem.

Acknowledgements

We would like to thank the professor Pierre Schaus*, our supervisor, for his time and patience during this academic year. This master thesis would have not be possible without his careful advices and his helpful guidance.*

We would also like to express our gratitude to our friends for some insightful discussions about different questionings in our thesis.

Finally, we would like to thank Etienne de Wergifosse *for the rereading of our thesis.*

Contents

Chapter 1

Introduction

> **Computer vision** is a field that includes methods for acquiring, processing, analyzing, and understanding images and, in general, high-dimensional data from the real world in order to produce numerical or symbolic information, e.g., in the forms of decisions.
>
> *Wikipedia*

Humans, as soon as they were born, begin a long learning procedure during all their life. A part of this learning consists in understanding the world in which they live. Our brain, our intellectual unit, continuously receives images perceived by our eyes and try to understand them. When trained enough, it becomes easy to distinguish and identify different objects from what we see. More than that, we are able to name the entities we detect from the perceived environment.

For instance, if we see a pair of shoes, we can say "Hey those are shoes!". The day after, if we see a pair of shoes in another environment, we are able to detect them and either say "Hey these are the shoes I saw yesterday" or "This is another pair of shoes compared to the one I saw yesterday". More than detecting and recognizing objects, the human brain understands the concept behind those objects. In fact the different pairs of shoes perceived are only instances of the concept "*a pair of shoes*". This concept can realize itself in a picture, in a drawing or in the real word.

Computer vision techniques aim at being able to achieve the same understanding of images as humans but with a computer. Unfortunately, for a computer, it is much more complicated to understand images, i.e, an array of pixels, and researchers have dedicated years of study in this area since the 1960's.

At that time, the first algorithm attempts dealt with *perceptrons* trying to automatically classify some images in one or another category after a learning step. But the results were very poor.

During the following years, some image processing techniques involving low level pixels computations emerged. These techniques are today the basic features of any image processing software. Things like resizing, blurring, noise reduction, contrasting and other filters are all basic functionalities in softwares like *Adobe Lightroom* or *Adobe Photoshop*.

But it is only from the 90's that the revolution in computer vision came out. With the democratisation of personal computers and the rapid improvements of computational power, new techniques of computer visions have emerged. SIFT, SURF or GLOH are all features description algorithms that can be used to describe and then recognize objects in images. At the same time the first optical characters recognition algorithm was born.

This democratisation of the computer vision field led to the development of applications in that domain. In the industry for example, robots exist that can understand and deal with their environment. We are also able to develop autonomous cars that are, in particular, able to follow the lines on the highway. Some surveillance softwares are able to detect and recognize humans faces, or compare and match fingerprints or DNA structures. It has become possible to automatically recognize licence plates with surveillance cameras on the highway. In fact, a lot of useful and diverse applications emerged with the improvement of computer vision techniques.

In this thesis, we focus on the detection, recognition and resolution of problems that can be described by a graph. One typical example is the sudoku problem. In this work, we explain the different steps that must be followed by an application to get a final solution from a picture containing the problem.

The figure 1.1 describes the general flow followed to solve the problem embedded in an input raw image. The first step consists in generating a standardized version of the input by applying image preprocessing techniques. Afterwards, the major step is to localize the problem in this standardized image. Finally, the data are extracted and processed to solve the problem leading to the final result.

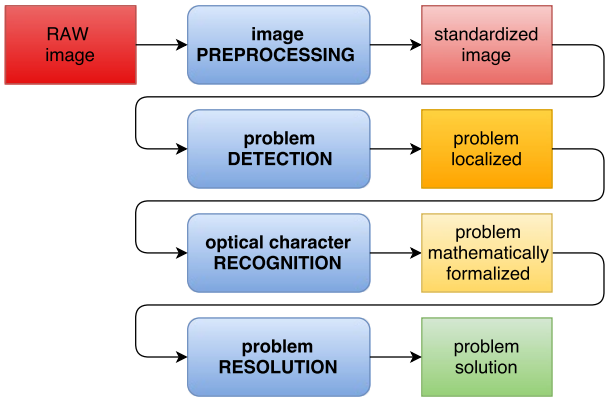


Figure 1.1: Execution flow of a problem recognition and solver algorithm

Our work is divided and organized in different chapters following this general flow.

First of all, a *state of the art* is established in the chapter 2. This is a very important chapter that presents the different techniques already existing in the study field of this thesis, i.e, solving problems by taking pictures. The existing technologies are reviewed and analysed to see in which manner it is similar or not to what is realized in this work. We also observe the way it can be reused or adapted in our case. It creates a comparison point for the work established in this thesis.

Then, the different steps of the developed algorithms are explained. First, in chapter 3, we explain what is a digital image and how it can be preprocessed. We also emphasize the importance of applying such preprocessing techniques to the raw image captured by the camera before proceeding to the detection of the problem.

The two next chapters, namely chapter 4 and chapter 5 present different approaches of problem detection. In the chapter 4, a problem oriented approach is developed. This dedicated approach is indeed specialized in the recognition of sudoku's. Dedicated approaches are powerful in the detection of one particular problem but can not be easily adapted for another problem.

In chapter 5, some general and modular approaches making use of graph matching techniques are presented. The aim of the approaches is to be able to detect any problem that can be described using a graphical representation. These techniques try to detect a given graphical representation of the problem in the input image. To achieve that, the first step consists in extracting a graph from the input image, the search space and then perform a matching between the two graphs. Solving this problem is in fact very complex (NP-complete) and that's why some good approximations techniques are presented in this chapter. The two first ones are *greedy-like* approaches. The first approach is coordinates dependant and uses the spatial informations of the graphs. The second is even more general and only relies on the nodes and links between them. That two latter approaches are *generative* : they try to progressively build the graph starting from a starting basic matching. Finally, a third approach has been considered, a *constraint programming* approach. Doing constraint programming to optimize a matching seems to be natural. Unfortunately, good formulations are hard to find and only small instances can be solved using constraint programming.

Once the problem structure of the problem is detected, the data of the problem must be recognized before being solved. In the chapter 6, we present how the data of the problem can be extracted to obtain a problem mathematically formalized and how to solve it.

Finally the chapter 7 presents some experimentations and evaluations of the different techniques explained during this work. Experiences are made in order to select the parameters for the preprocessing techniques and the techniques are compared in terms of accuracy and efficiency.

Before beginning, let's first briefly define some notations that are equivalently used

in the remaining of the thesis :

- The desired graph described by the user in the input of the algorithm is called *model graph*, *mGraph* or *model*.
- The graph extracted from the input image is called *search space graph*, *sGraph*, *search graph*, *space graph* or *target graph*.
- A node or an edge part of the *model graph* are respectively called *model node* or *mNode* and *model edge* or *mEdge*.
- A node or an edge part of the *search graph* are respectively called *search node*, *target node* or *sNode* and *search edge*, *target edge* or *sEdge*.

Chapter 2

State of the Art

In this chapter, a state of the art regarding the available technologies relative to our problematic, i.e solving problems by taking pictures, is proposed. It is important to collect and analyse in which manners the existing technologies are similar or different regarding the objective sought. Similar methods may be reused and adapted to solve the problematic faced in this work. It is also important to understand whether an existing technology can fit with our problem or why it may be irrelevant. Finally, we present the differences and or improvements of our work compared to the existing ones and why it may be more relevant in our application field.

2.1 Dedicated approaches

In terms of dedicated approaches, a lot of different applications exist and perform well in their specific domains. In this section, for curiosity, some examples of existing applications are cited.

On the Google Play Store, you can find the application *Sudoku Vision*. Specialized in the sudoku resolution, this application is very similar to our dedicated approach presented in the chapter 4.

On the App Store, an application called *MathPix*, just released on May 2016, allows its user to handwrite an equation on a sheet of paper, take a picture of it with his iPhone and the application gives the solution.

On the website *AirBnB*, during the sign up procedure, an identification proof is required. The user can provide either a scan or a picture of his ID card and the website will automatically extract the needed information.

2.2 Graph matching approaches

Graphs are widely used in computer sciences. A common data structure used is the *attributed relational graph* (ARG). It is a graph in which nodes and edges bear additional information about the data represented. ARGs are more expressive, flexible and powerful than vector representation like features vectors. In computer vision and pattern recognition, such graph structures are widely used and different techniques exist to extract graphs from images. In this thesis, the focus is set on graph matching techniques applied once a graph has been extracted from the input image. In this section we will review some techniques more or less similar to the ones developed in this work and explain why it is or not applicable to the particular problem of graph matching we face.

For years now, the problem of *graph isomorphism* has always been at the heart of the research in computer engineering. This problem, that has neither been demonstrated to be solvable in polynomial time nor to be NP-complete, is simply stated :

Given two graphs, the problem is to determine if they are isomorphic (=essentially the same), i.e if there exists a bijection between the nodes and edges of each graph.

Recently, in 2016, László Babai [1], proposed a new algorithm to solve this problem in quasi-polynomial time $exp(\log n^{\mathcal{O}(1)})$ while the previous best bound was $exp\left(\mathcal{O}(\sqrt{n \log n})\right)$.

Unfortunately this promising result can't be applied for the purpose of our work because it is too restrictive. Indeed, the *sGraph* extracted from the image may be noisy and contain too many nodes and edges but can also contain too few nodes and edges if the graph detection does not perform well.

Finding the isomorphim of a model graph *mGraph* into a larger target graph *sGraph* is known as *subgraph isomorphism*. This problem, less restrictive than the strict graph isomorphism, has been proved to be NP-complete [3][7].

In 1976, Ullmnan [24] worked on a recursive backtracking procedure to solve this problem. The running time remains exponential in the most general case but when fixing the choice of the search graph *sGraph*, it was shown to take polynomial time. Furthermore, when the model graph *mGraph* is planar and *sGraph* is still fixed, the running time is reduced to linear time. In 2010, Ullmann substantially updated his 1976 algorithm [25].

In [10], the important notion of *graph labelling* or *node signature* is described. The principle consists in assigning to each node a *label* depending on its neighbourhood. For example, the label can be the degree of the node. When considering the matching of the model node *mNode* with the target node *sNode*, comparing the label of each node is a relevant factor to evaluate if they can possibly match. Indeed, in the subgraph isomorphism problem, if the degree of *sNode* is less than the degree of *mNode*, they never can match together. This notion of *graph labelling* is widely used to perform pruning in the tree search of such matching algorithms.

In [4], a subgraph isomorphism procedure is proposed using topological node fea-

ture(=label) constraints. The labelling procedure of the node makes use of the *degree*, the *clustering coefficient* (= number of edges between adjacent nodes), the *number of cliques of size k* and the *number of walks of length k passing through a particular node.*

In [34], a constraint programming approach is proposed to solve the sugraph isomorphim problem as a constraint satisfaction problem (CSP). The pruning is also achieved thanks to a node labelling procedure allowing to define a partial order on the labels.

[19] is another constraint programming approach for the subgraph isomorphism.

These algorithms are one step closer to what we seek to do but still doesn't stick exactly to our problematic. Indeed, here, a perfect isomorphism between the model graph *mGraph* and a subgraph of the target graph *sGraph* is still needed. However, in our study case, the *sGraph* extracted from the image is often imperfect because affected by noise. In the best case, we hope the extracted graph will exactly contains the model graph but it is a very strong assumption on which we can't rely. Nevertheless, this kind of algorithms pointed out a very important notion that we will use in the further proposed matching algorithms : the *importance of a good node labelling* to improve the pruning.

The problem of identifying a subgraph from a search space graph *sGraph* that approximately corresponds to a model graph *mGraph* is known as *approximate subgraph isomorphism.* This problem relaxes even more the *subgraph isomorphism* problem and becomes even more costly to solve. The relaxation allows the presence of additional noisy nodes or edges as well as the lack of some nodes or edges.

In [32] and [33], a constraint programming approach is used to solve the approximate subgraph isomorphism problem. It makes use of a labelling of the nodes and allows a part of the graph to be approximated while the remaining part has to be isomorphically matched. The problem is considered as a constraint satisfaction problem (CSP) and is satisfied as soon as the required part of the graph is isomorphically matched.

The *approximate subgraph isomorphism* is close to what we want to achieve. Nevertheless, it is still too restrictive. In our case, we don't know which nodes and edges from the model graph may be missing into the target graph. Each node and edge can eventually be missing and what we want is to minimize the number of missing nodes and edges so that the matching found is the largest and covers at best the model graph. Using the previously cited approach in our study case is impracticable since there is no mandatory nodes or edges that must be present and the pruning relies on that mandatory part of the graph.

Our problem could be defined as an *approximate subgraph matching optimization problem* where we want to identify a sugraph from the search space graph *sGraph* that best matches with the model graph *mGraph.* It is an optimization problem and a complete solution requires an exhaustive exploration of the search tree which is impracticable since the problem is *NP-complete.*

In [31], a tool for approximate large graph matching (called *TALE*) is proposed. It also relies on a careful labelling of the nodes to perform a good pruning. Because a complete exploration of the search is impossible, a clever idea is introduced here. An analysis of the model graph is performed to *find particular nodes that may be good starting nodes* where the search should begin. Then the search starts from the starting nodes in a greedy-like scheme. Using the labelling and the starting nodes heuristics gives good and fast results in large graphs.

2.3 Contributions

In this thesis, two main ideas have been retained from these papers :

- The *labelling of the nodes* is efficient to reduce the domain of each node and increase the pruning.
- A clever analysis of the model nodes is a good idea to *find good starting nodes* from where the search should begin.

In this work, two main graph matching algorithms are presented, both of them making use of these two ideas. The first one is coordinates dependant and implants the coordinates of the nodes as a new node information in the search. In the graph matching procedure, it is interesting to observe how this information can guide the search and efficiently performs the approximate matching. Our second approach is similar to the approach presented in [31] but is tuned to perform approximate subgraph matching relative to problems identification.

2.4 Alternatives approaches

Graph matching is a way to retrieve a model structure into an image. However, other applications relying on different procedures exist. In this section, some of them are briefly presented.

Some realizes face recognition with the help of ARG's. The goal being again to match two ARGs together, one describing a face model, the other being the sample where to identify the model. In [13], a grid shaped graph is built on the model face to be identified. Nodes are labeled with features that precisely describe the local gray-level distribution. Edges describe the relative position of vertices. If an elastic distortion of the graph is detected in a sample, the model ARG is identified as well as the corresponding face. The model graph structure can be improved, as shown in [28]. The nodes gathering information focus on specific facial landmarks to establish a more relevant correspondence.

Other applications use principal component analysis (PCA) to solve recognition problems [15]. Variables are introduced to describe the interesting features of the images.

The goal is to describe with principal components, a set of linearly uncorrelated variables, the space observed. If the variables of the space are strongly correlated, the use of the intrinsic dimensionality given by PCA will reduce the data needed to efficiently describe the whole space. PCA can be used to represent pictures of human faces [23][30] or for a fast computation of the matching error to speed up the retrieval of corresponding ARGs [29].

Chapter 3

Image processing techniques

In this first technical chapter of this thesis some *image processing techniques* are presented. We will see and understand that before applying some image recognition algorithms to detect and then solve the problem, it is essential to *preprocess* the images taken in input. On the figure 3.1, the first step of the whole algorithm is highlighted. It consists in the preprocessing of the input image.

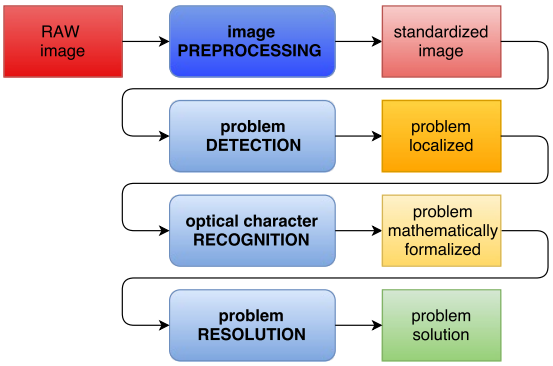


Figure 3.1: Execution flow of the algorithm - Preprocessing

In this chapter, different techniques of *resizing*, *smoothing*, *thresholding* and *thinning* will be explored and understood. Some of these techniques are then retained for the preprocessing step of the final algorithm.

3.1 What's a digital image?

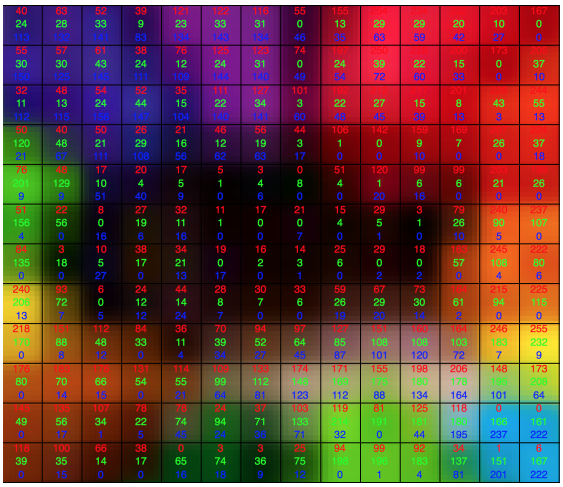
So the first step before digging into the understanding of the image recognition algorithms is the preprocessing of the input image. And to be able to apply some image processing techniques, it's important to understand in what consists a digital image.

While a human can perceive shapes and objects included in a given environment when looking at a picture, from a computer point of view, it's a bit different.

A digital image is in fact a two dimensional grid of *pixels*. The more pixels an image contains, the more information it contains and then possibly the more details it has[1]. We can thus evaluate the quality of a image by its *pixel resolution*, i.e the number of pixels it has (generally given in mega-pixels Mpx). Because images are rectangular, the *resolution* is the multiplication of the number of pixels on its *height* times the number of pixels on its *width*.

As stated an image is made of pixels. A *pixel* is the smallest constituent of the image and can be described in a *color system*. The most usual one is the *RGB color model* for Red-Green-Blue color model. In that model, a pixel is described by a triplet (r, g, b) where the r, g and b values, ranging from 0 to 255[2], indicate the amount of red, green and blue included in that pixel. The pixel $(0, 0, 0)$ is black while the pixel $(255, 255, 255)$ is white. The image on the figure 3.2 shows an example of a digital image aside from a zoom on the pixels composing it.

(a) A image in its full resolution



(b) RGB values of the pixels when zooming on the image

Figure 3.2: Example of a digital image [18]

Knowing how a computer handles a digital image, we understand that, unlike a human, it is not trivial for a computer to detect objects or shapes among a grid of pixels. We'll now see some pre-processing techniques useful to apply to an image before

[1]The blurred version of a non-blurred image has less details than its non-blurred version despite having the same amount of pixels.

[2]8-bits value

executing recognition techniques on its pixels. For illustrative purpose, a recent camera which shoots pictures of $24Mpx$ will result in a uncompressed raw coloured picture of $24 \cdot 10^6 \cdot 3 \cdot 8 = 576Mbits = 72MB$.

3.2 Resizing

When dealing with digital images, we often need to enlarge or reduce their dimensions. This process is called *resizing*. During this process the *destination picture* will have more or less pixels than the *source picture*. A *mapping procedure* is thus required to attribute to each destination pixel, one or a combination of pixels from the source picture. The basic mapping scheme is defined as

$$Map : \mathbb{N}^2 \to \mathbb{N}^2 : (x_d, y_d) \to (x_m, y_m)$$

which maps a destination pixel coordinate (x_d, y_d) to a mapped source pixel coordinates (x_m, y_m) where to find the pixel value. When the coordinates mapping are known, the value of the destination pixel denoted by $P'[(x_d, y_d)]$ is equal to the value of the source pixel denoted by $P[Map\,((x_d, y_d))]$

According to the OpenCV API [17], three interesting mapping procedures exist. We'll briefly talk about the *nearest-neighbour interpolation*, the *bilinear interpolation* and the *area interpolation*.

3.2.1 The need of resizing

Before exploring the different mapping procedures, let's understand why the resizing of the image is an important step in the preprocessing phase. In input of the algorithm, images taken by the user will be received, theoretically taken by its smartphone camera but not necessarily. The incoming pictures can then have a bunch of different pixels resolution.

The first argument for the resizing is to *standardize the size* of the input image. It is very important because the following preprocessing techniques that will be applied rely on some thresholds parameters whose results depend on the size of the image. In order to simplify the choice of the thresholds to a fixed value independent from the size of the image, a standardized image size is preferred.

The second reason why we would like to resize the images is to *reduce the future computation cost*. Indeed, we need to keep in mind that, later, some image recognition algorithms will be applied. And these algorithms work by analysing the image, i.e the pixels of the image. Obviously, the more pixels the image has, the more computations the algorithm needs to perform. While the cameras on smartphones have sensors ranging approximately from 6 to 20 megapixels, common image recognition algorithms don't

need pictures of more than 1 or 2 megapixels to work just fine. It is then a common practice to reduce the size of the input image to such a pixel resolution.

3.2.2 Nearest Neighbour Interpolation

It is the easiest way to interpolate. The principle is to take the closest neighbour from the source image after a linear transformation of the destination coordinates to fit with the source image. The mapping function transforming a source image of size $h_s \times w_s$ to a destination image of size $h_d \times w_d$ is defined as

$$Map\left((x_d, y_d)\right) = \left(\overbrace{round\left(\frac{x_d \cdot w_s}{w_d}\right)}^{x_m}, \overbrace{round\left(\frac{y_d \cdot h_s}{h_d}\right)}^{y_m} \right)$$

The value of each destination pixel $P'[(x_d, y_d)]$ is taken from the value of the source pixel computed by this mapping procedure and is equal to $P[(x_m, y_m)]$

For instance, let's consider the resizing of 6×6 image into a 2×2 image. The four pixels of the destination image can then be computed as $P'(1,1) = P\left(\frac{1\times6}{2}, \frac{1\times6}{2}\right) = P(3,3)$, $P'(1,2) = P(3,6)$, $P'(2,1) = P(6,3)$ and $P'(2,2) = P(6,6)$

3.2.3 Bilinear Interpolation

The nearest neighbour interpolation can be improved by considering all the four pixels surrounding the unrounded pixel from destination image back to the original image. Indeed let's consider the two pixels coordinates $(2.01, 2.01)$ and $(2.49, 2.49)$ before being rounded by the mapping procedure. It's obvious that $(2.01, 2.01)$ must be rounded and mapped to $(2, 2)$ but what about $(2.49, 2.49)$? It will also be rounded and mapped to $(2, 2)$ while it is nearly as close to $(2, 3)$, as $(3, 2)$ or $(3, 3)$. It is from this observation that the bilinear interpolation came out. With this interpolation scheme, the pixel value of (x_d, y_d) is computed as the linear weighted sum of its four neighbours. For instance

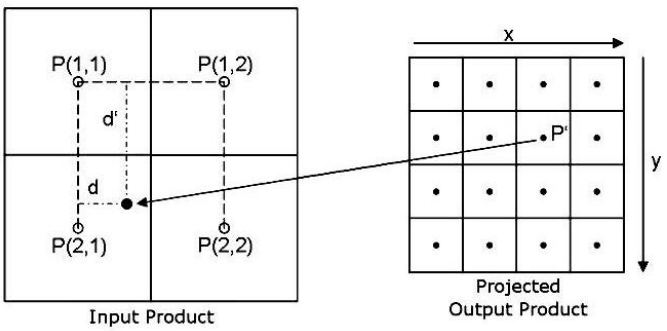


Figure 3.3: Example of pixel value computation for the bilinear interpolation [8]

for the example on figure 3.3, the value of the destination pixel will be [8] :

$$\begin{aligned} P'(x_d, y_d) = \quad & P(1,1) \cdot (1-d)(1-d') + P(1,2) \cdot d(1-d') \\ & + P(2,1) \cdot d'(1-d) + P(2,2) \cdot d \cdot d' \end{aligned}$$

where d and d' are the absolute values of the horizontal and vertical component of the vector starting from the upper left neighbour pixel of the unrounded theoretical pixel (here $(1,1)$) to the theoretical unrounded pixel coordinates (see figure 3.3). In that way, the computation of $P'(x_d, y_d)$ is the weighted sum of its four surrounded pixels values.

3.2.4 Area Interpolation

The complete description of area interpolation is beyond the scope of this thesis. Nevertheless the principle is to place the unrounded mapped pixel over the source image and make an averaged sum on the covered pixels. This is interesting when shrinking images where a destination pixel will cover several pixels when mapped to the source image. For instance, when shrinking an image by dividing its original resolution by a factor 3 (from $24Mpx$ to $8Mpx$ for example), each destination pixel would cover 3 pixels from the source image and its value would result in the weighted sum over that covered pixels.

3.3 Smoothing

The process of *smoothing* an image can be achieved through different techniques. The most common one, called *blur*, are described in [2] and [11]. The different applications where a blur can be useful depend of the desired objective. It can simply be for the design effect of a blurred image, or to reduce a certain type of noise embedded in the image or also to simplify it in terms of details, to make it less sharpened. To preserved the sharpness while reducing the noise, other smoothing procedures may be used such as *bilateral filtering*.

In this section, we'll talk about the *mean blur*, the *gaussian blur* and the *median blur*. These are the most widely used blur operations. Afterwards, *bilateral filtering*, is described. In the section 7.1.1, adequate smoothing procedures and parameters are selected for our application.

3.3.1 The kernel concept

Most of blurring methods make use of what is called a *kernel*. As described in [11], a *kernel* can be seen as a *sliding window* (a two dimensional array of values) "flying" over the image (an array of pixels) and computing the new value of each pixel when passing over it. The output image is the *convolution* of the input image with the given kernel.

To understand how it works, consider the figure 3.4 which explicitely illustrates the computation of 2 arbitrary destination pixels. These destination pixels are the convolution of the kernel with the relative source pixels.

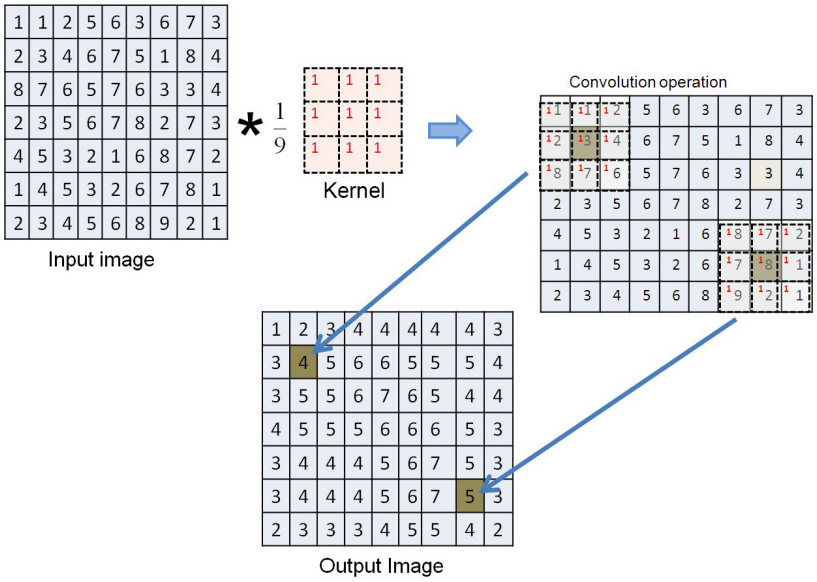


Figure 3.4: Convolution of the kernel with the input image. [26]

In the context of blurring an image, a square kernel with an odd side length is used. Some examples of dimension are 3×3 or 5×5. Furthermore, the sum of each element in the kernel must equal 1. Indeed, each new pixel is nothing else but a weighted sum of the pixels in the neighbourhood, some of them having, possibly, a higher weight than others. To be able to compare to effect of different kernel sizes when applying a blur, it is also important to always have the same image resolution in input.

3.3.2 The mean blur

The *mean blur*, also called *simple blur*, is the simplest blurring scheme. For a kernel of side n, each element will have the value $\frac{1}{n^2}$ which is, in English, the mean of the surrounding pixels. For example, for $n = 5$, we have

$$\text{kernel}_{Mean} = \frac{1}{25} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

The image 3.4 uses a mean kernel of size 3×3 to illustrate the convolution operation.

3.3.3 The Gaussian blur

The *mean blur* is easily understandable and straightforward to apply. But this technique is not very fair since it gives the same importance to every pixel of the kernel. The centred pixel, for which the new value is being computed during the convolution has indeed the same importance than further neighbouring pixels. Would it not be possible to change the kernel in order to make it more equitable ? It is in fact easily fixed by giving an higher value to the center point of the kernel and by attributing less importance to the neighbouring pixels. The Gaussian blur uses the Gaussian function to accomplish the computation of the kernel values. This function is maximal on its center and decrease the further we walk away from this center. Of course this new approach is more costly in terms of computations but gives better results. The 2D Gaussian function centered in $(0, 0)$, with an amplitude of 1 and a standard deviation $\sigma_x = \sigma_y = \sigma$ is :

$$f(x, y) = e^{-\left(\frac{x^2+y^2}{\sigma}\right)}$$

The choice of the parameters is arbitrary. The only thing to remember is to normalize the kernel so that the sum is equal to 1. For instance a 5×5 Gaussian kernel could be

$$\text{kernel}_{Gaussian} = \frac{1}{273}\begin{pmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{pmatrix}$$

3.3.4 The median blur

The *median blur* does not make use of the kernel convolution as the two previous methods do. Instead, the principle is to look for the *median pixel value* among a given neighbourhood around the considered pixel. This blurring technique is particularly used for the *reduction of salt and pepper noise*.

3.3.5 Bilateral filter

OpenCV uses the bilateral filtering described in [6] which relies on [22]. We briefly describe the algorithm. Bilateral filtering is a simple, non-iterative scheme for edge-preserving smoothing. The goal is to get rid of the imperfections of the image while conserving the edges, i.e the pixels where the image brightness or colour change sharply.

The idea is to realize in the *range* of an image what traditional filters do in its *domain*. In *domain* filtering, like the *gaussian* procedures, two pixels are close to each other if they occupy nearby spatial locations. Therefore, the influence of pixels on each other decays as the distance falls off. Similarly, in *range* filters, pixels are considered close when they are similar in terms of their colours or intensities. Pixels which are very dissimilar will almost have no influence on each other. The combination of *range* and

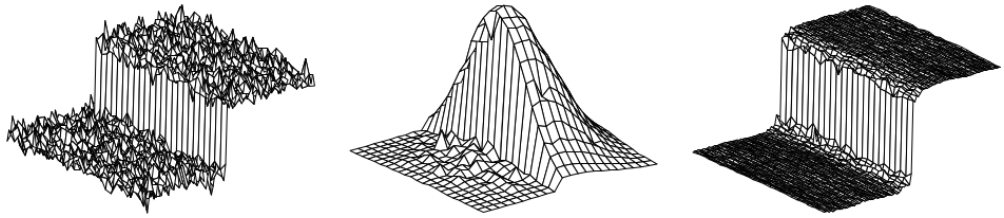


Figure 3.5: Application of bilateral filtering on noised edge between bright and dark region. (a) Step perturbed with Gaussian noise. (b) Similarity function for a 23 x 23 filter support centered 2 pixels to the right . (c) Filtered edge for $\sigma_r = 50$ gray levels and $\sigma_d = 5$ pixels.

domain filtering is called *bilateral filtering*.

Figure 3.5 (a) from [22] shows noised sharp boundary between dark and bright region. The function evaluating the similarity of a single pixel is shown on (b). Normalization ensures that the weights for all pixels add up to one. The filter replaces the bright (resp. dark) pixel at the centre of the kernel by an average of the bright (resp. dark) pixels in its neighbourhood while essentially ignoring the dark (resp. bright) pixels. The bilateral filtering result is shown on (c). Thanks to the range component of the filter, crisp edges are preserved. At the same time, thanks to the domain component, a good filtering behaviour is achieved.

3.3.5.1 Parameters

The bilateral filter is controlled by two parameters : the range and spatial parameters σ_r and σ_s. Their respective effects are evaluated in the section 7.1.1. The *spatial parameter* σ_s tunes the influence of distant pixels. The *range parameter* σ_r weights the influence of pixels with different intensity values. The bilateral filter becomes closer to Gaussian blur as the range parameter increases i.e. the influence of different intensities decreases. Note that, since the spatial and range weighting are multiplied, if one get close to 0, no smoothing occurs.

3.3.6 Adaptive bilateral filter

The adaptive bilateral filter (ABF) is an improvement of the classical one. It is described in [35]. As the bilateral filter, the adaptive technique is able to smooth the noise, while strengthening edges and textures in the image. The improvement is that the parameters are optimized with a training procedure. The resulting images of the ABF procedure are significantly sharper than the ones produced by the bilateral filter.

3.4 Thresholding

When working with images in computer vision, it is rare to need the three color channels that offers an RGB coloured image coming from the camera sensor of a smartphone. What is important for image recognition algorithms is to be able to distinguish shapes, detect edges, corners, identify embedded text, and so on. Most of the time, the color is irrelevant.

This is where *thresholding techniques* become interesting. It consists in converting each pixel of the coloured image into a *black or white pixel*. The thresholding function looks like

$$\text{Thresh} : \mathbb{D}_{RGB} \to \mathbb{D}_{BW} : P_{RGB} \to P_{BW}$$

where $\mathbb{D}_{RGB} = \{(r, g, b) \mid r, g, b, \in [0, 255]\}$ is the domain of the coloured pixels and $\mathbb{D}_{BW} = \{(0, 0, 0), (255, 255, 255)\}$ is the domain of the black or white pixel. The aim of the threshold function is thus to simplify the image since we pass from a domain size of $|\mathbb{D}_{RGB}| = 256^3$ into a domain size of $|\mathbb{D}_{BW}| = 2$.

In order to apply this threshold, most techniques first convert the coloured image into a *grayscale image* following the basic formula

$$(r, g, b) \to \left(\frac{r+g+b}{3}, \frac{r+g+b}{3}, \frac{r+g+b}{3}\right)$$

We'll now present two thresholding techniques : the *binary threshold* and the *adaptive threshold* as described in [2] and [11].

3.4.1 Binary threshold

The *binary threshold* function is very simple. For each pixel, it attributes either a black color B or a white color W according an arbitrary threshold value $T \in [0, 255]$. So according a grayscale pixel value P_{GS} in input[3], we have

$$\text{Thresh}_{binary}(P_{GS}) = \begin{cases} = W & \text{if } P_{GS} > T \\ = B & \text{if } P_{GS} <= T \end{cases}$$

This binary threshold scheme is simple and the results strongly depends on the arbitrary choice of the threshold T. Some threshold choices can be

- $T = 127.5$ which is the middle of the domain but as it doesn't take into account the luminosity condition of the image, it gives poor results and isn't used in practice.
- $T =$ **mean** which is the mean of the pixel values but doesn't take into account the different luminosity conditions in different parts of the images.

[3]For simplicity, since we only deal with uni-color channel pixel, we compare the grayscale pixel value P as if it was a natural value while it is in fact a triplet (p, p, p)

- $T =$ **median** which is the median of the pixel values. It is imperfect because we don't always want the exact half of pixels to be white and the other half to be black.

3.4.2 Adaptive threshold

We rapidly see the limitations of the binary threshold. None of the common choices is very good and even if we can find a good threshold value T by trials/errors, this value T won't work with other pictures shot in other conditions. A clever approach needs to be found.

The *adaptive threshold* is one smarter way to threshold images. The principle, based on the binary threshold, is the following : for each pixel P, only consider its neighbourhood, i.e a small frame of its nearest pixels and compute the threshold value T according to the surrounding pixels in the frame (it can be the mean, the median or a even a Gaussian weighted sum of the values); then attribute the value to the pixel P according to a binary threshold with threshold T.

This approach is much more efficient since it is more general and behaves well with different luminosity conditions on the same picture. The arbitrary parameter is the size of the frame which depends on the image resolution. But for a given resolution, it is reasonable to assign it to a fixed experimentally determined value.

A small improvement is the possibility to set a constant C to a value that will be subtracted from the initial threshold value T to give a new threshold value T'. This can be used to favour a brighter output image.

3.5 Thinning

Thinning is the process of reducing the width of every *line* contained in a binary image (the *pattern*) to a one pixel width. The goal by achieving thinning may be to reduce the cost of further processes or the time required to perform a shape analysis. Therefore, thinning algorithms are used in a variety of applications such as document analysis or pattern recognition [12].

In this section we detail two popular thinning algorithms. Both are *iterative* thinning algorithms. *Iterative* because the deletion decision for an individual pixel relies on the results of the previous iteration(s). This deletion decision also relies of the neighbourhood pixels. At the end, a 1-pixel thick *skeleton* is obtained. The two algorithms presented here differ by the quality of the obtained skeleton, the number of iterations or subiterations and the preservation of the connectivity. They are compared and evaluated in the section 7.1.4.

3.5.1 Zhang-Suen thinning

T.Y. Zhang and C. Y. Suen proposed a *Fast Parallel Thinning Algorithm* in 1984 [36]. It is based on a thresholded image defined by a matrix of binary pixels[4]. The principle of this algorithm is to reassign new values to each pixel during a series of iterations.

The neighbourhood of a pixel P1 = (i,j) is defined and designated as shown in Table 3.1. The new value given to a pixel at the nth iteration depends on its own value and its neighbourhood at the $n-1$th iteration. Therefore all pixels can be processed simultaneously. The method consists in removing all the adjacent points that do not belong to the skeleton. Each iteration is divided in two subiterations in order to preserve the connectivity of the skeleton.

$P_9\ (i-1, j-1)$	$P_2\ (i, j-1)$	$P_3\ (i+1, j-1)$
$P_8\ (i-1, j)$	$P_1\ (i, j)$	$P_4\ (i+1, j)$
$P_7\ (i-1, j+1)$	$P_6\ (i, j+1)$	$P_5\ (i+1, j+1)$

Table 3.1: Designation of the neighborhood of $P_1 = (i, j)$

In the first subiteration, the pixel P_1 is deleted from the pattern if it satisfies the following conditions :

(a.) $2 \leq B(P_1) \leq 6$

(b.) $A(P_1) = 1$

(c.) $P_2 * P_4 * P_6 = 0$

(d.) $P_4 * P_6 * P_8 = 0$

where $A(P_1)$ is the number of 01 patterns[5] in the ordered list P_2, P_3, ..., P_9 composed of the neighbours of P_1. $B(P_1)$ is the number of non zero neighbours of P_1. It can be shown that this iteration removes only the south-east boundary and north-west corner of the pattern which does not belong to an ideal skeleton.

The second subiteration is similar to the first one and is applied on the pixel P1 just after the first subiteration. The pixel P1 is deleted if the same conditions (a.) and (b.) are fulfilled in the same time as the conditions (c'.) and (d'.) which slightly change compared with before :

[4]Each pixel has either the value 1 or 0

[5]When looking at a list, a *01 pattern* appears if two successive elements in the list have values 0 and 1.

(c'.) $P_2 * P_4 * P_8 = 0$

(d'.) $P_2 * P_6 * P_8 = 0$

As for the first iteration, this second iteration removes only the north-west boundary and south-east corner of the pattern which do not belong to an ideal skeleton.

The results of those iterations are shown on the subfigures (a) and (b) of 3.6. Once no more pixel can be removed, the final skeleton is obtained as shown on (c).

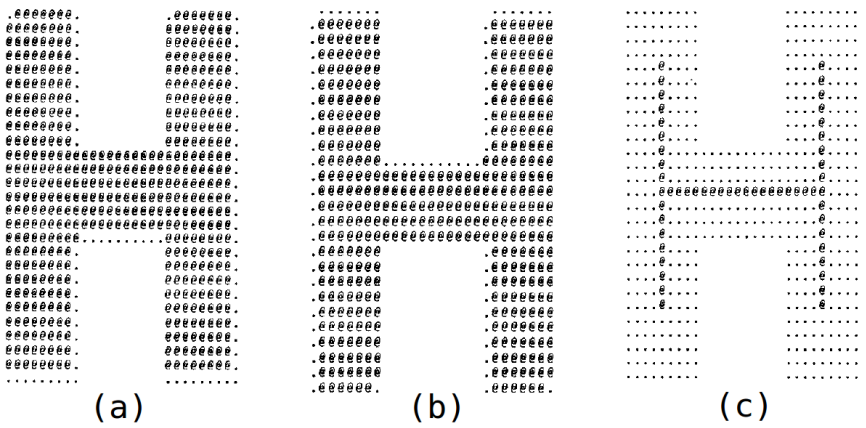


Figure 3.6: Zhuang-Suen : two first subiterations and final result

3.5.2 Guo-Hall thinning

In 1989, Zichang Guo and Richard W. Hall proposed a *Parallel thinning with two sub-iteration algorithms* [9]. Using the notation described in table 3.1, one can define :

$$N(P_1) = \min(N1(P_1), N2(P_1))$$

$$N1(P_1) = (P_2 \vee P_3) + (P_4 \vee P_5) + (P_6 \vee P_7) + (P_8 \vee P_9)$$

$$N2(P_1) = (P_3 \vee P_4) + (P_5 \vee P_6) + (P_7 \vee P_8) + (P_9 \vee P_1)$$

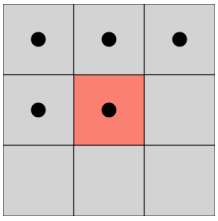
A point will be deleted if it satisfies :

(a.) $C(P_1) = 1$

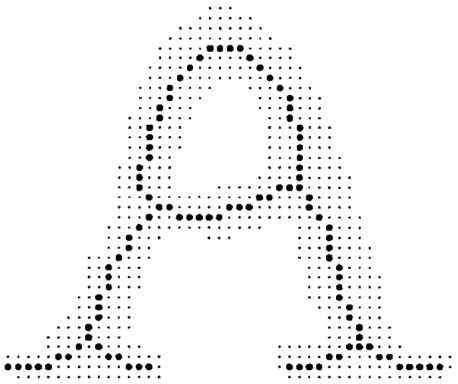
(b.) $2 \leq N(P_1) \leq 3$

(c.) $\begin{cases} (P_2 \vee P_3 \vee \overline{P_5}) * P_4 = 0 & \text{if in an odd iteration} \\ (P_6 \vee P_7 \vee \overline{P_8}) * P_8 = 0 & \text{if in an even iteration} \end{cases}$

where $\vee$ is the logic OR operation and $C(P)$ is the number of distinct eight-connected components of 1's in P's neighbourhood. $C(P) = 1$ ensures local connectivity, an example is shown on figure 3.7a. This combined with (b.) and (c.) means that there is only one group of 8-connected 1's around P. The deletion of P will therefore not break the connectivity. Criterion (c.) for odd iteration deletes north-east pixels while (c.) in even iterations removes south-west pixels. The result of Guo-Hall thinning applied on a pattern representing the letter A is shown on figure 3.7b.



(a) Example of pixels with one 8-connected components of ones ($C(P) = 1$)



(b) Guo-Hall result on a pattern representing the letter A

Figure 3.7: Guo-Hall thinning

Chapter 4

A problem oriented approach

4.1 The choice of the problem

The first step of our study has been to observe and understand some techniques already existing for the detection and the resolution of some basic problems. As stated in chapter 2, interesting applications already solving the *sudoku problem* exist.

This puzzle seemed to us an alluring one to start a first approach. Indeed a sudoku is nothing but a square grid of 9 by 9 cells, some of them containing a number. But from a recognition algorithm point of view, the problem isn't that simple. The aim of a recognition algorithm is to detect sudoku's in real world pictures, sudoku's that can be printed in a newspaper, possibly framed by another box or standing next to another sudoku. With such a consideration, detecting a straight line inside our input image may or may not be part of the sudoku. On another hand, the way to think to a sudoku as a grid of 9 by 9 cells or as being 10 parallel lines perpendicularly intersecting 10 others parallel lines will also result in a different implementation of the recognition algorithm.

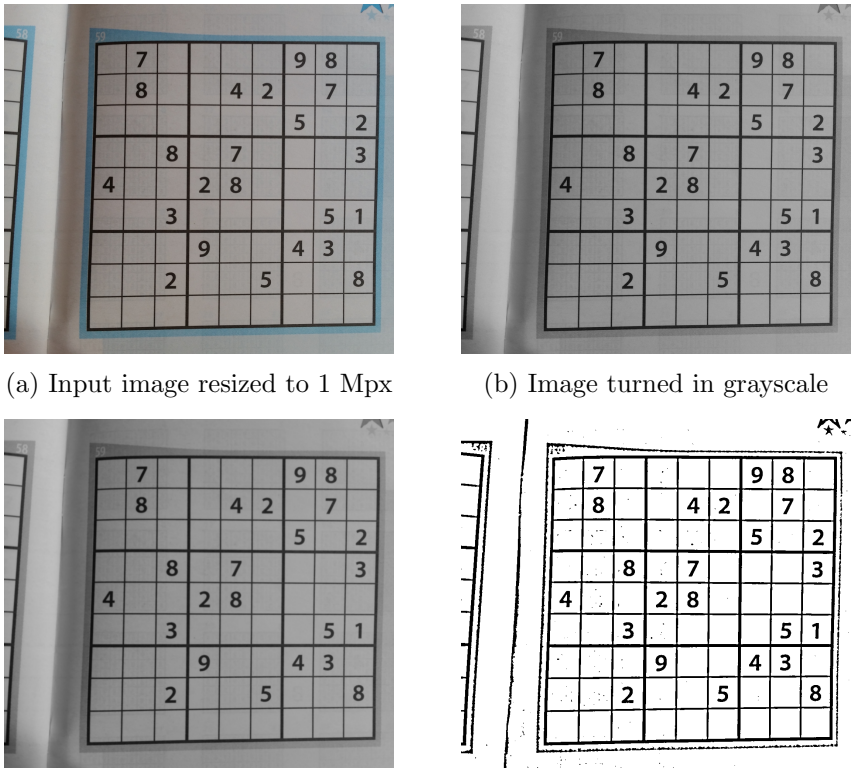
With all that observations, we decided to develop a *sudoku oriented* recognition algorithm. This first approach is *problem oriented*, which means that we know what we are looking for and that we'll use all the properties and particularities of the sudoku to detect it. It also means that the algorithm will be specialized for the detection of sudoku's and won't be easily adapted for the detection of other problems. Nevertheless, the process of detection will be faster than a more general algorithm.

In this chapter, we first explain the preprocessing techniques retained and applied to the input image. After that the focus is put on the detection of the sudoku. It is the second step of the whole algorithm and certainly the most important one.

4.2 Preprocessing

In the chapter 3, we developed a lot of different image processing techniques. In this section, we present the techniques that we retained after a careful study[1] and that we apply to preprocess the input images before being able to proceed to the *problem detection* itself.

First of all, the input image is loaded and *resized* to $1Mpx$ using the *area interpolation* technique. The next thing to do is to threshold the image. The image is turned into *grayscale* and to facilitate the thresholding task, a *bilateral filter* with parameters $\sigma_s = 5$ and $\sigma_r = 80$ is applied. It smooths the image by removing irrelevant details. Once done, it's time to *threshold the image* using the adaptive threshold procedure with a kernel size of 19 and a subtracting factor of 5.



(a) Input image resized to 1 Mpx

(b) Image turned in grayscale

(c) Image smoothed with a bilateral filter

(d) Thresholded image

Figure 4.1: Illustration of the different steps of the preprocessing phase of the algorithm

Using the basic preprocessing scheme described here gives good results to begin the detection part of the algorithm. For illustration purposes, have a look at the figure 4.1.

[1] See chapter 7 for more details

As can be observed, the thresholded image gives a very good starting point to detect the sudoku.

4.3 Detection

In this section, we talk about the operations that need to be applied to the standardized image output by the preprocessing step of the algorithm. The aim of these operations is to detect the sudoku on the image before passing it to the optical character recognition phase which will become aware of the data of the sudoku. As a reminder, have a look at the figure 4.2 which highlights the *detection step*.

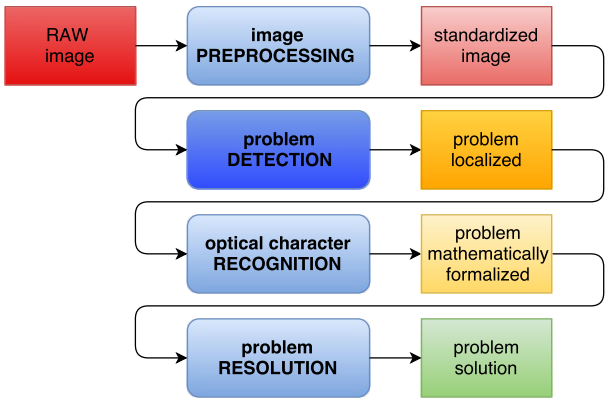


Figure 4.2: Execution flow of the algorithm - Detection

The topology of the sudoku is pretty Cartesian, it is composed of 81 square cells distributed among 9 bigger squares forming the big square grid. The goal is to be able to detect that square grid, i.e the sudoku. To be able to detect that grid in the image, some assumptions have to be made and it is important to think to the borderline cases.

Here are the assumptions that we made and the borderline cases we must deal with :

- The sudoku is the main part of the picture, it should take more than half of the width and more than the half of the height, which is 25% of the image surface.
- The sudoku must be correctly oriented on the image :
  - The rotation with respect to an horizontal line should lies inside $[-40°, +40°]$.
  - The sudoku should be shot in front of the sensor, the perspective should not deteriorate the sharpness of the sudoku
- The sudoku must be complete, not cropped or torn.
- The sudoku must be drawn in a dark color on a light background.

```
input :      image -img
output :     set of polygon -P
init :       -P an empty set
function :   getPolygons

  C ← findContours(img)
  for each c in C :
    polygon ← approxPolyDP(c)
    if polygon has between 4 and 8 sides : graph
      P.add(polygon)
  P ← keep the largest polygons #the 5 best for ex.
  P ← P sorted by number of edges
  return P
```

Pseudo-code 4.1: Detecting polygons - pseudo-code

- Only one sudoku should appear in the picture.
- The grid can be surrounded by another larger rectangle.
- A part of another sudoku can appear next to the sodoku to detect.
- The sudoku can be shot in different lighting conditions (problem partially dealt with by the preprocessing).
- The sudoku can be located anywhere in the picture, in the upper left corner or the lower right corner.
- Since the sudoku can be captured in a newspaper or a book, it can be a little skewed.

4.3.1 Detecting the polygons

According to the topology of the sudoku, it is obvious that squares, especially large ones are relevant feature that need to be detected. Because of the real world conditions of our application and the assumptions spotted out above, we cannot be sure that the biggest square found on the picture will be the sudoku, neither that it will be a square. If it is a little skewed, it can be a pentagon or an hexagon for example.

So, the goal of this step is to *detect a set of relevant polygons* that may contain the sudoku. The pseudo-code 4.1 sketches how to obtain that set of polygons. Only the polygons having of degree 4 and 8 are retained[2]. It makes use of two functions `findContours` and `approxPolyDP`, part of the OpenCV library and respectively described in [20] and [16]. We'll now explain how these two functions work.

[2]We call *degree* of a polygon, its number of sides

`findContours`

Let's first explain how the function `findContours` works. The explanation here is based on [20] where additional information can be found.

The principle is to scan a binary image (for convenience, we denote the black pixel by the value 1 and a white pixel by the value 0), i.e an array of 0 or 1 using the same scheme as a TV raster[3]. The scanning aims at detecting the*1-component*s (a connected area of black pixels) and the *0-component holes* (a connected area of white pixels) with their respective borders and hierarchy. An example is shown in figure 4.3.

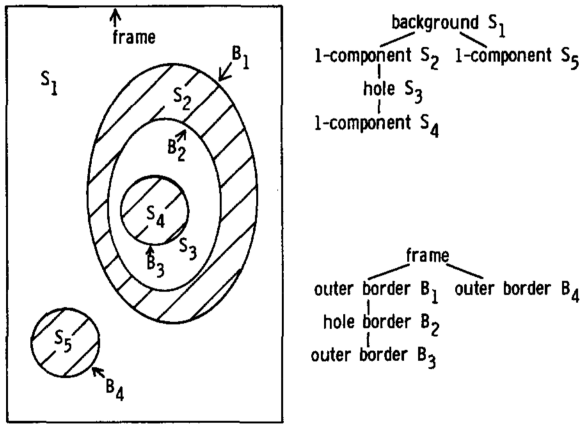


Figure 4.3: Topological structure of a binary image [20]

The raster scans the array of pixels and stops momentarily when it encounters a pixel (i, j) which is either an outer border ($(i, j) = 1$ and $(i - 1, j) = 0$) or a hole border ($(i, j) \geq 1$ and $(i + 1, j) = 0$)[4]. When such a pixel (i, j) is found, a *border following procedure* is executed in order to mark each pixel of the border with a unique value (the black pixels value changes) according to the marking policy :

- Mark the current the pixel (p, q) with the value $-UIN$[5] if the current border we are following (remember that a border passes between pixels) lies between the pixels $(p + 1, q)$ and (p, q) and that the pixel $(p + 1, q)$ is part of a 0-component and that the pixel (p, q) is part of 1-component.

- Otherwise mark the current (p, q) pixel to UIN if (p, q) is not yet marked by another already followed border.

[3]Scanning the array line by line from left to right
[4]A pixel having a value greater than 1 is a black pixel already detected in another border, see below.
[5]UIN refers to a unique identification number

Once the marking procedure is done, the raster scan resumes. To illustrate the functioning of the marking strategy, let's have a look to the figure 4.4. The circled pixels are the ones on which the raster scan stops to execute the marking procedure.

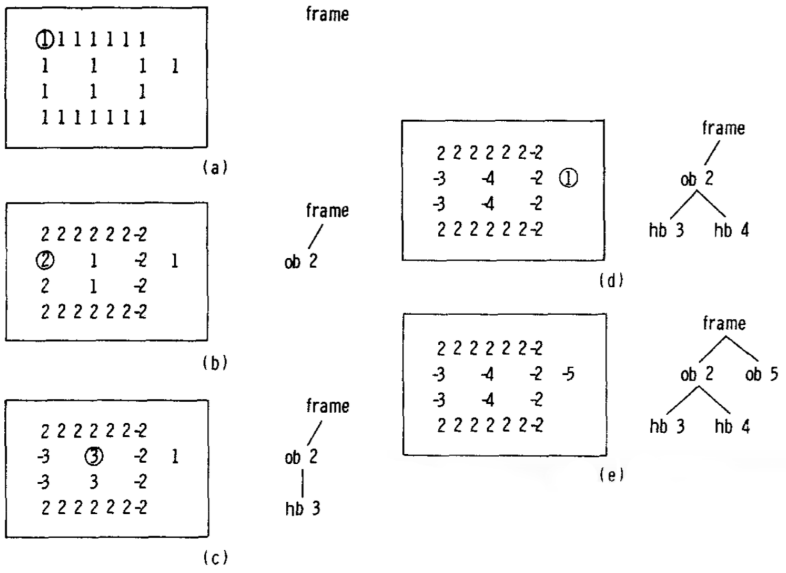


Figure 4.4: Illustration of the marking policy detecting the contours of the image [20]

`approxPolyDP`

Once the set of contours have been detected, we look at the extraction of polygonal approximation of these contours. Given a contour, i.e a ordered list of pixels composing it, the principle is pretty straightforward. An example is illustrated on the figure 4.5. This example is generic for curves but is the same for closed contours which is what interests us here.

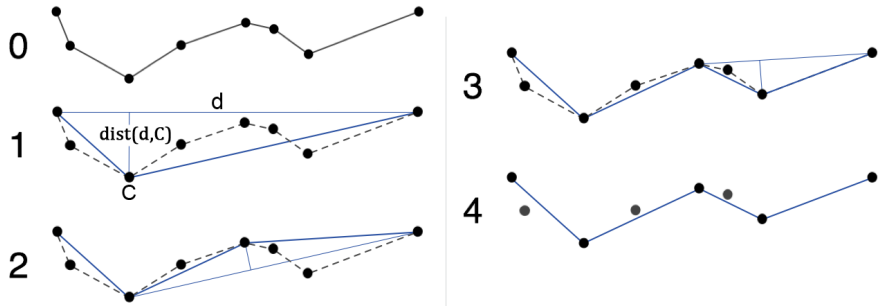


Figure 4.5: Illustration of the Ramer-Douglas-Peucker algorithm used by the function `approxPolyDP` [16]

The aim of the algorithm is to simplify the curve composed of a lot of points. To do that, the algorithm begins by approximating the curve by a straight line d starting from the first point of the curve and ending at the last point of the curve (possibly the same point for closed contour) (see step 1). It then finds the further point from that line, denoted by C. Then

- Either $dist(d, C) < \epsilon$, where ϵ is a tolerance criterion selected by the user in pixels, and then the algorithm stops
- Or $dist(d, C) \geq \epsilon$ in which case the curve d is updated to include the point C in its passing points and the procedure starts again by finding the new further point from the curve.

Finally the contour has been approximated by a polygonal curve passing by a subset of points of the initial contour.

Illustration

When the `getPolygons` function finishes, a set of the 5 largest polygons is returned. Let's observe what is returned for a sudoku screened by that function. As can be seen on the figure 4.6, good polygons are detected, including the sudoku grid in purple. Because the original image is a little skewed, the square grid is approximated by an hexagon by the `approxPolyDP` function.

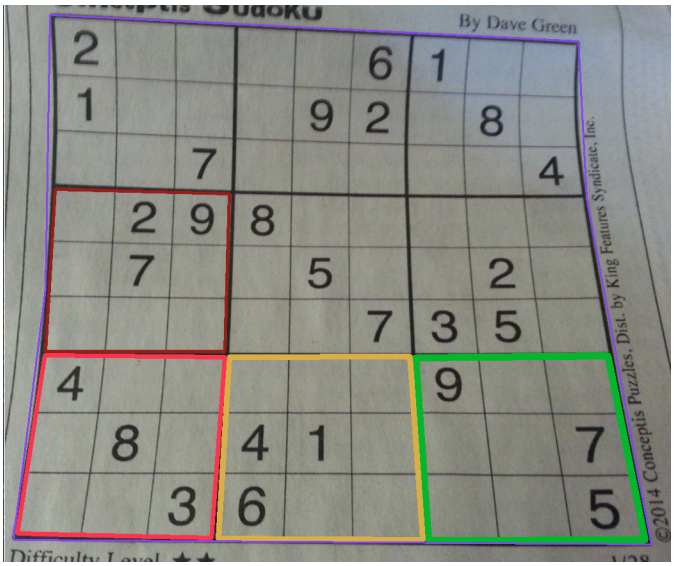


Figure 4.6: Five best polygons found by the `getPolygons` function

```
input :     a polygon -p, the thresholded image -img
output :    the remapped part of the image based on the polygon -
   remap
init :      -destMapping is the list of coordinates of the 4
   corners of the future remapped picture (500x500)
function :  remap

  p ← extractSquare(p)
  if p is null :
    return null

  sourceMapping ← getOrderedPoints(p)
  perspective ← getPerspectiveTransform(destMapping, sourceMapping
   )
  remap ← warpPerspective(img, perspective)
  return remap
```

Pseudo-code 4.2: Remapping a polygon

4.3.2 Remapping

When the polygons have been detected, each of them standing for a potential sudoku grid, the next thing to do is to *remap each polygon* into a square as if it had been drawn in two dimensions. Indeed, due to the perspective, it is important to remap the sudoku into a perfect square before passing the remapped square to the next step which will check if that square is really a sudoku.

To remap the polygon, two policies have been chosen :

1. We only want to remap polygon of degree 4. Polygon's degree higher than 4 are filtered to only keep 4 most suitable vertices for a square.
2. The previous decision assumes that the skewing imperfections can be ignored for the remapping. Only the perspective is corrected during the remapping.

The remapping procedure is sketched on pseudo-code 4.2. It makes use of sub functions that will now be clarified.

`extractSquare`

Given a polygon in input, the aim of the function is to extract, if possible, four vertices that will most likely represent a rectangle. Because the polygon's degree in input will be at most 8, a naive approach, that generates every possible combinations of 4 vertices among the given set of vertices which compose the polygon, is reasonable. Indeed, it will have at most $\frac{8!}{4!4!} = 70$ combinations generated.

Each of the generated combination is checked to see if it roughly represents a square[6], tolerances being allowed on the measures. Finally, among all the possible squares, only the one with the largest area is chosen and returned, if any.

`getOrderedPoints`

This function simply reorganizes the 4 corners of the square in the following order : the upper left, the lower left, the lower right and the upper right. It is important to organize the points to correctly map the square points to the desired destination mapping.

`getPerspectiveTransform`

Given four pairs of corresponding coordinates in 2D, the aim of this function, implemented in OpenCV, is to *determine a perspective transformation matrix* T. Despite considering 2D coordinates, the perspective transform deals with 3D coordinates because the transformation quietly zooms in or zooms out the picture when applied. To goal is then to find the matrix T such that

$$K_i \begin{pmatrix} x'_i \\ y'_i \\ 1 \end{pmatrix} = \begin{pmatrix} T_{xx} & T_{xy} & T_{xz} \\ T_{yx} & T_{yy} & T_{yz} \\ T_{zx} & T_{zy} & T_{zz} \end{pmatrix} \begin{pmatrix} x_i \\ y_i \\ 1 \end{pmatrix}$$

where K_i stands for the third dimension and i ranges from 0 to 3 for the four pairs of known mapped points. In this equation, the point (x_i, y_i) is mapped to (x'_i, y'_i). With the fours pairs of mappings, we have a system of $9 + 4 = 13$ parameters with only $4 * 3$ equations. To solve this system, T_{zz} can be arbitrary constrained to 1 in order to fix the scaling factor of the matrix T. The solution of the system, now properly defined, gives the perspective transform matrix T.

`warpPerspective`

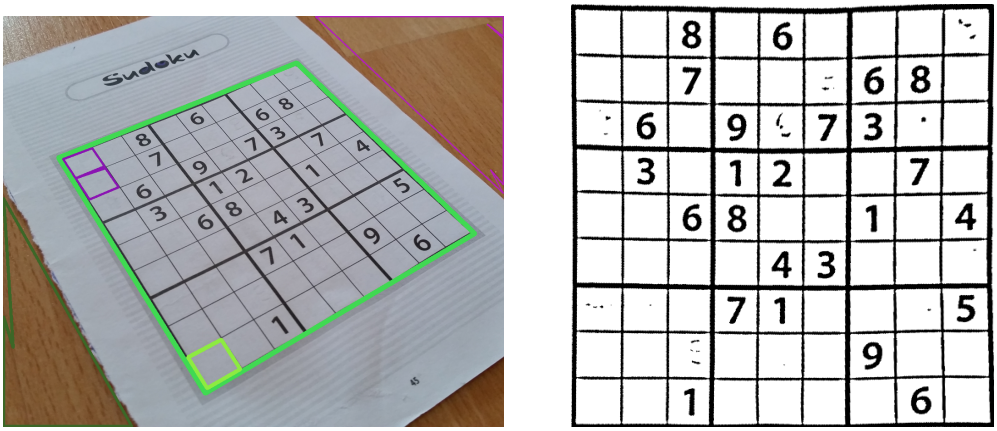
Implemented in OpenCV and described in [17], this function simply applies the perspective it receives to extract and remap the desired zone from the input image it receives. The dimension of the output image has been fixed to 500×500. The relevant destination coordinates (x', y') are computed as follows :

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \frac{1}{T_{zx}x + T_{zy}y + T_{zz}} \begin{pmatrix} T_{xx}x + T_{xy}y + T_{xz} \\ T_{yx}x + T_{yy}y + T_{yz} \end{pmatrix}$$

Illustration

The remap procedure aims at extracting and remap a potential sudoku grid into a two dimensional square. An illustration of the remapping of a correctly detected polygon can be seen on the figure 4.7.

[6] Checks are made on the sides length and angles



(a) Original image with the polygons detected

(b) Polygon containing the sudoku remapped into a 2D square

Figure 4.7: Illustration of the execution of the remapping procedure

```
input :     the remapped grid -remap
output :    the set of lines -allLines
function :  getSudoku

  linesH, linesV ← getLines(remap)
  if not isSudokuFound(linesH, linesV) :
    return null

  extrapolateLines(linesH)
  extrapolateLines(linesV)
  return linesH ∪ linesV
```

Pseudo-code 4.3: Extracting the lines of the sudoku

4.3.3 Extracting the lines

Now that a polygon potentially containing the sudoku has been remapped into a 2D square, it is time to check if it is indeed a sudoku. The fastest and easiest way to do that is to look for 10 horizontal equidistant parallel lines and the same for vertical lines. If we can find that inside the remapped image, it is highly probable that the sudoku is correctly detected.

The pseudo-code 4.3 gives the intuition on how the sudoku can be extracted from the remapped figure. Some helpful functions are used in this pseudo-code. They are detailed below.

`getLines`

When the remapped portion of the image where the sudoku can be is available, it is required to ensure that it is indeed the sudoku that is enclosed inside. Because the sudoku grid is made of 10 horizontal and vertical lines, it is a good idea to *look for lines inside the remapped image*. This is exactly what the `getLines` function does.

Detecting straight lines inside a picture can be made with what is called the *Hough Line Transform*. Described in [2] or in [11] and implemented in OpenCV, the principle is the following. Any line in an image can be mathematically represented by the polar equation

$$r = x\sin(\theta) + y\cos(\theta)$$

where r is the distance from the origin (the upper left corner of the picture) and θ is the rotation angle with respect to the x axis (see figure 4.8).

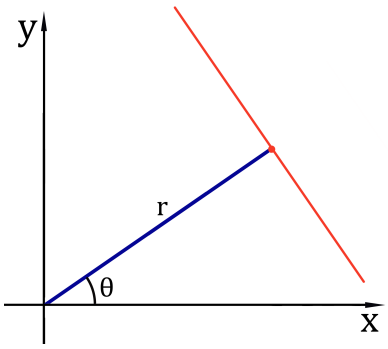


Figure 4.8: Line represented in polar coordinates

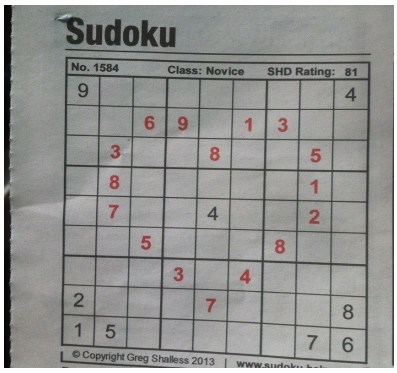
The principle of the *Hough Line Transform* is to consider every possible line, i.e every possible (r, θ) couple and then proceed to a vote. Each pixel is scanned and when a black pixel with coordinates (x_i, y_i) is encountered, it votes for each considered line (r_j, θ_j) respecting the equation $r_j = x_i\sin(\theta_j) + y_i\cos(\theta_j)$ (with a small error tolerance). At the end of the vote, all the couples (r_j, θ_j) having a satisfactory number of votes will be considered as lines[7]. Of course the step between two successive r considered and between two successive θ considered is configurable. In our case, we set $r_{step} = 1$ pixel and $\theta_{step} = 1°$, which, for the $500x500$ remap image in input, considers $500 \cdot 180$ possible lines and scans all the $500 \cdot 500$ pixels. For that dimension, we fixed the satisfactory number of votes to 300. For an image of longest dimension L, the Hough Line Transform has a complexity $\mathcal{O}(180L^3) = \mathcal{O}(L^3)$.

The Hough Line Transform will find a set of lines L. This set can contain diagonal lines and redundant lines due to the thickness of the sudoku grid. Before returning the

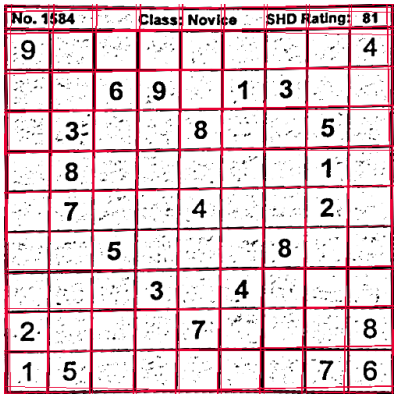
[7]The satisfactory number of votes depends on the length of the line we are looking for and on the resolution of the image.

set of line L, it is required to filter it to only keep horizontal and vertical lines. The set also needs to be cleared to remove redundant lines. Finally in certain cases, a final check must be made to remove useless lines, (part of box surrounding the sudoku for example). And only then, a set of lines will be relevant and can be returned.

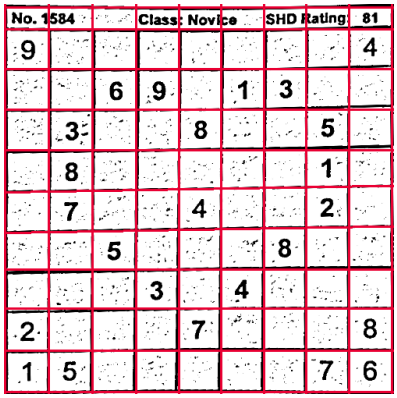
Consider figure 4.9 that illustrates the work of this function.



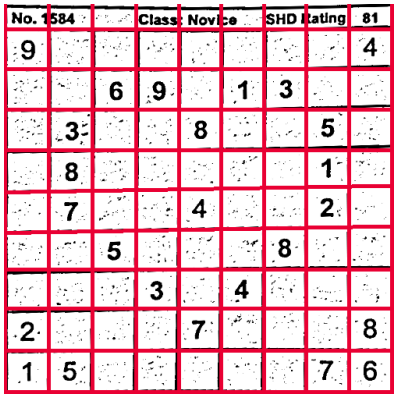
(a) Original picture



(b) All lines detected by the Hough Transform



(c) Diagonal lines filtered out as well as redundant lines



(d) Remaining useless lines filtered out - top line removed

Figure 4.9: Illustration of the work of the function `getLines`

`isSudokuFound`

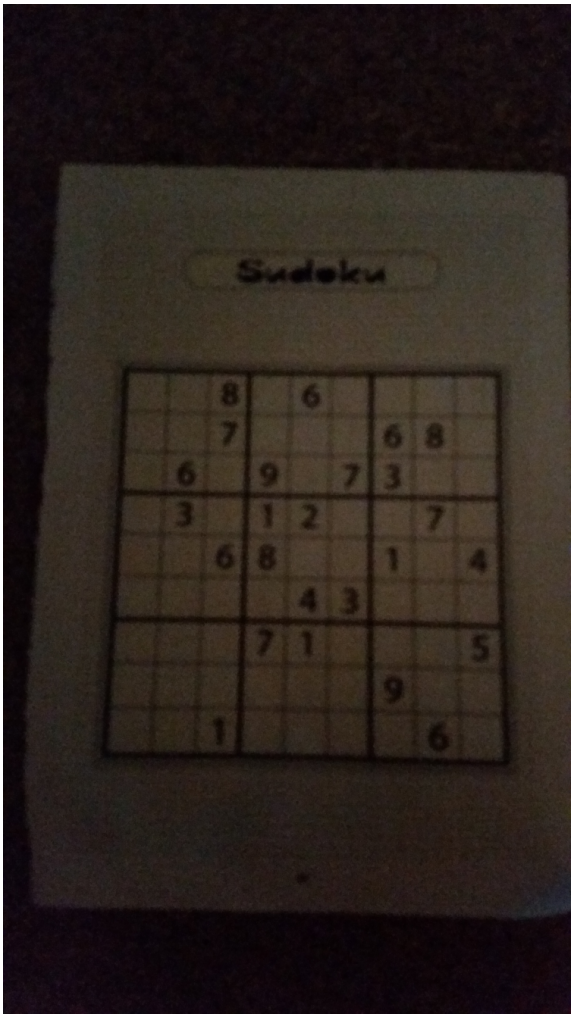
After the execution of the function `getLines`, the sudoku is considered to be found if, for both direction, either

- more than 5 lines have been found or
- the 4 *main hard lines* delimiting the 9 squares have been found.

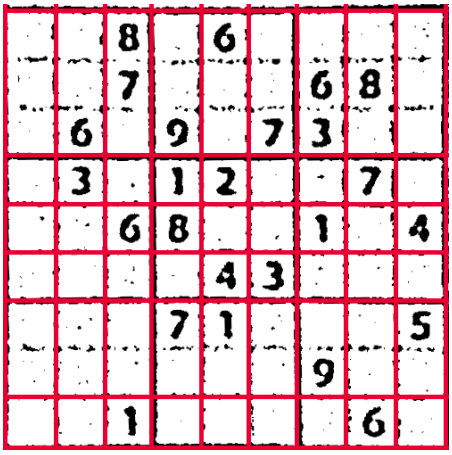
And this is exactly what this function checks.

`extrapolateLines`

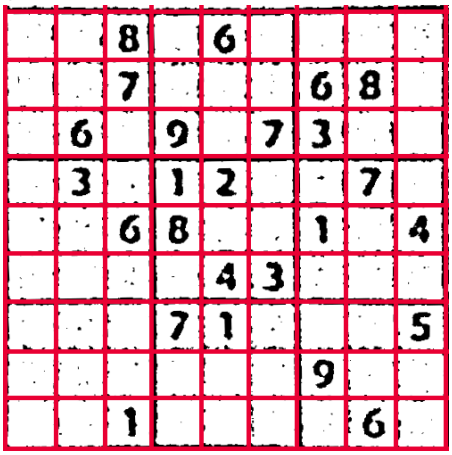
If less than the 10 horizontal and vertical lines have been found, it is possible to extrapolate the missing lines because we know that they are equally distanced. This is the role of the `extrapolateLines` function which is applied on the set of horizontal lines and vertical lines. Once done, the parent function `getSudoku` returns the union of the two extrapolated sets. The figure 4.10 illustrates such a specific case.



(a) Original picture



(b) Lines kept at the end of the `getLines` functions



(c) Horizontal missing lines have been extrapolated

Figure 4.10: Illustration of the extrapolation part of the function `getSudoku`

4.4 Summary of the approach

Let's now *summarize the problem oriented approach* studied during this chapter. As said, the approach is problem oriented, and the problem that has been selected is a sudoku. This chapter briefly explained the preprocessing techniques applied to the input before focusing on the detection of the sudoku itself. In this section, the key points of the approach are summarized.

When receiving an image from the user, it's important to *standardize and simplify* the input image through the process of **preprocessing**.

1. The image will be *resized* to a pixel resolution of $1Mpx$.
2. Colours being irrelevant for most computer vision applications, the image is *turned in grayscale*,
3. *smoothed with a Gaussian blur* before
4. being *thresholded in black and white* with a adaptive threshold method.

Once preprocessed, it's time to proceed to the **detection** of the sudoku itself.

1. The 5 most relevant *polygons are extracted* from the thresholded image. This is realized by finding the contours in the image and then by making a polygonal approximation each of them.
2. Each polygon potentially contains the sudoku square grid. Before looking for the sudoku lines inside the polygons, it is necessary to *remap it into a 2D square*. If the polygon has more than 4 sides, this step begins by extracting the enclosing square, then it computes the perspective mapping matrix and finally proceed to the remapping.
3. The final step of the detection consists in *detecting the lines* inside the remapping to ensure it is indeed a sudoku. It uses the Hough Transform to detect all the lines, then filters the diagonal lines, the redundant ones and the not equally spaced ones if too many were found. Finally if too few were found, missing lines are extrapolated.

Chapter 5

Graph matchings approaches

In the previous chapter, a problem specific approach has been developed. But what if the next time, it is not a sudoku but a *mathematical pyramid* like the one shown in the figure 5.1 that we would like to recognize. Well, the whole algorithm should be adapted to detect such a new problem. The insight of this chapter is to develop some new more general approaches that could rely on a simple description of the desired problem to detect it. This description would simply be supplied in the input of the algorithm. With such an approach, if the problem to detect changes, the modifications to make are pretty straightforward.

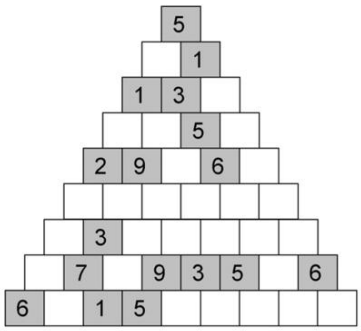


Figure 5.1: Another problem - the mathematical pyramid

In this second part of our study, the focus has been put on the detection of the problem thanks to a *graphical representation*. A simple observation led to this approach : if a problem can be defined by a specific graph structure that is unlikely to appear accidentally, then finding such a graph structure in the image is strictly equivalent to finding the problem. For example, in the sudoku problem, finding a graph corresponding to the grid is enough to find the problem's data. For the latter problem, the graph could be composed of the 100 nodes at the intersections of the vertical and horizontal lines of

the grid. And the edges would simply link the adjacent nodes together.

In this chapter, the two main tasks of the problem detection approach are explained :

- The first one is to *extract a graph structure* from the picture. A graph structure is composed of nodes and edges. To achieve that, some image preprocessing techniques are applied to the image to extract its skeleton. Then dedicated algorithms analyse this skeleton to extract some nodes and detect the edges linking them together. This is the purpose of the section 5.1.
- The next step is to identify in this network the structure of a predefined problem, itself described by a graph. The aim is thus to *find a matching* making the correspondence from the *model nodes* to the *space nodes* and from the *model edges* to the *space edges*. Different matching techniques are presented in this chapter. In the section 5.2, a *coordinates dependant search* is presented. In the section 5.3, a generalization is made to become *coordinates independent*. Finally, an optimization approach using constraint programming is described in the section 5.4.

The *graph matching problem* covered in the second step can be formalized as follow. Given a model graph $mGraph$ and search space graph $sGraph$, find $\overline{mGraph} \subset mGraph$ such that

$$\nexists\, \overline{mGraph}' \subset mGraph : \text{valOf}(\text{matching}(\overline{mGraph}')) > \text{valOf}(\text{matching}(\overline{mGraph}))$$

where `matching` is a procedure finding the best matching of the graph passed in argument to a subgraph of $sGraph$ and `valOf` evaluates the value of the matching.

Because the extracted $sGraph$ is imperfect and noisy, it is unlikely to be able to exactly find all the nodes and edges from the $mGraph$ inside the $sGraph$. In the $sGraph$, some noisy nodes and edges can appear while some other nodes and edges can also be missing. That's why, the aim of the matching procedure is to find the largest subgraph $\overline{mGraph} \subset mGraph$ which is called *approximate subgraph matching optimization* and not exactly this $mGraph$ into a subgraph of $sGraph$ which is called *approximate subgraph isomorphism*.

We first study a trivial algorithm to determine whether $sGraph$ (of n nodes) contains a copy of $mGraph$ (of k nodes) [27]. Let $\{m_1, ...m_k\}$ be the nodes of $mGraph$. Then, we look at all the n^k ordered tuples of k nodes of $sGraph$ $\{s_1, ...s_k\}$. For every edges (m_i, m_j) in the model graph, we check if the corresponding edges (s_i, s_j) exists. If it is true for all i, j, $sGraph$ contains a copy of the graph $mGraph$. The running time of a such algorithm is $\mathcal{O}(k^2 n^k)$.

When dealing with *NP problems*, it is required to *find good heuristics and relaxations* to

- *Strongly prune* the search space with bounds computations and not lose time exploring irrelevant parts of the search tree
- *Choose good variable-selection and value-selection heuristics* to improve the pruning and guide the search to a solution.
- *Relax the problem* to accept a good solution but not necessary the best one.
- *Only explore some parts of the search space* in a smart way to find good approximations of the best solution.

In this chapter, the search schemes presented make use of a combination of some of these heuristics and relaxations. The different searches use different graph structures and makes assumptions on the topology of the graph in order to strongly reduce the search space size. It allows the search to apply to a reduced search space with a much more reasonable size. The *coordinates dependent search* and the *coordinates independent search* limits the search space size in a *greedy-like fashion*. Despite performing the search on a reduced search space, good results are observed in practice. The *CP approaches*, on the other hand apply on the whole search tree but diversify the search by use of a large neighbourhood search (LNS).

5.1 Graph extraction

The first step in the graph matching approach in thus to *extract a graph structure* from the image where the problem might be. This step is independent from the matching search so that any technique can be used to extract a graph structure from an image. Only the data type used must coincide with the matching search. In this section, we'll developed a way to extract a graph structure from an input image.

The graph extraction process reviewed here is inspired by the NEFI tool (Network Extraction From Images) [5]. NEFI combines algorithms following an *extraction pipeline* scheme : for each pipeline section, several interchangeable algorithms can be used. To extract a network from an image, the typical steps are *preprocessing*, *segmentation*[1], *graph detection* and *graph filtering*. The Figure 5.2 shows a flow chart illustration of the NEFI's pipeline. Results of some key steps are shown in figure C.1 available in Appendix C.1. The steps to obtain these results are now described.

5.1.1 Preprocessing

As discussed in chapter 3, the aim of the image preprocessing algorithms is to modify the source image to simplify it and only keep the required information for the following

[1]NEFI distinguishes the segmentation from the preprocessing because the segmentation is compulsory to be able to perform the graph detection. Nevertheless, the segmentation remains a preprocessing technique.

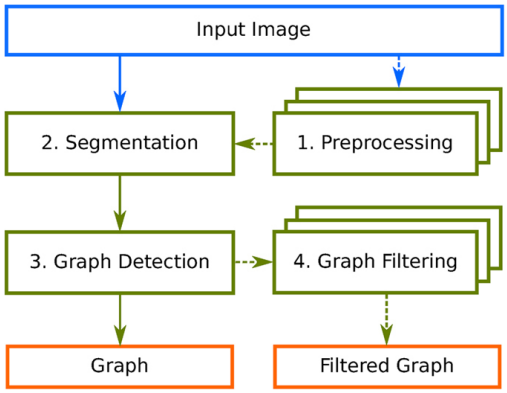


Figure 5.2: A flow chart illustrating NEFI's pipeline components in green boxes. Dashed arrows depict optional sections of the pipeline. Blue and orange boxes denote NEFI's input and possible outputs respectively. [5]

part, here the *segmentation*. Typical preprocessing algorithms are *resizing* and *smoothing* procedures.

This step is crucial since it impacts the following parts of the process. If the blur and noise reduction remove too many details, important features of the sought problem may be erased. Conversely, light blur and gentle noise reduction may lead to a foreground which still contains a lot of irrelevant components.

As already stated in the previous chapter, a study of the preprocessing procedure used is detailed in chapter 7 and the parameters retained are discussed.

5.1.2 Segmentation

The mandatory preprocessing technique is the *segmentation*. The aim of the segmentation is to separate the structure of interest containing the problem (*foreground*) from the rest of the image (*background*). Adaptive thresholding is one of the most efficient, yet simple, segmentation procedures. For those reasons, this procedure is the one retained for the segmentation part of the graph extraction. However, other algorithms like *guided watershed* or *GrabCut* can easily be swapped or combined with the basic tools to improve the segmentation.

Once again, a proper study of the result of the different segmentation parameters is realized in section 7.1.3.1.

5.1.3 Graph detection

The graph detection consists in a sequence of algorithms that detects the nodes and edges of the graph in a segmented image. This is performed in three successive steps: *thinning*, *node detection* and *edge calculation*.

5.1.3.1 Thinning

The first step consists in reducing the segmented foreground to a 1-pixel thick skeleton while preserving the connectivity properties of the image (for both the foreground and the background). Two different thinning algorithms can be used : the Zhang-Suen algorithm [36] and the Guo-Hall algorithm [9]. They are precisely described in the section 3.5 and evaluated in section 7.1.4. In our study, the Zhang-Suen algorithm has been retained.

5.1.3.2 Node detection

The second step aims at detecting the position of the nodes from the skeleton. An adaptation of the criteria used in the Zhang-Suen algorithm [36] is applied : a pixel that belongs to the foreground's skeleton is considered as a node if it is either the end of an edge or the intersection of at least three edges.

To find nodes in the image, the skeleton is analysed. In this binary image, the black pixels are part of the *background* and the white pixels are part of the *foreground*. To find the nodes, each pixel of the skeleton and their *1-neighbourhood*[2] pixels are analysed. For each white pixel of the skeleton, a clockwise walk through the 1-neighbourhood is performed. One counts the number of times a skeleton pixel appears next to a background pixel. That counter defines the *degree* of the skeleton pixel. If the count is 1, 3 or 4, the skeleton pixel is considered as a node with the same degree. Figure 5.3 illustrates the different cases increasing the *pixel degree*. If the count equals 2, the pixel can either belong to an edge or be a degree 2 node. A pixel on an edge should be considered as a node of degree 2 if the edge performs a sudden change of direction at this pixel. Those pixels are identified during the edge calculation using the *Ramer-Douglas-Peucker* algorithm. It is already used and described in the dedicated approach when using the `approxPolyDP` function (Section 4.3.1).

Since only four schemes can be satisfied at the same time, the maximal degree of a node is four. This is inevitable if the node are detected using the 1-neighborhood of the skeleton's foreground. Fortunately, nodes that are very close to each other can be merged afterwards to create higher degree nodes. This will be performed during the graph filtering step of the extraction pipeline.

5.1.3.3 Edge calculation

When the nodes have been discovered, it is time to extract the edges. Given the skeleton and the nodes, the edge computation is pretty straightforward. A search similar to a breadth first search (BFS) is performed starting simultaneously from every node detected before. An *identification number* and a *queue* are attributed to each simultaneous search.

[2]The *1-neighbourhood* of a pixel P are the 8 pixels surrounding that pixel P, i.e they are the pixels part of the 3x3 square centred in P without the pixel P itself.

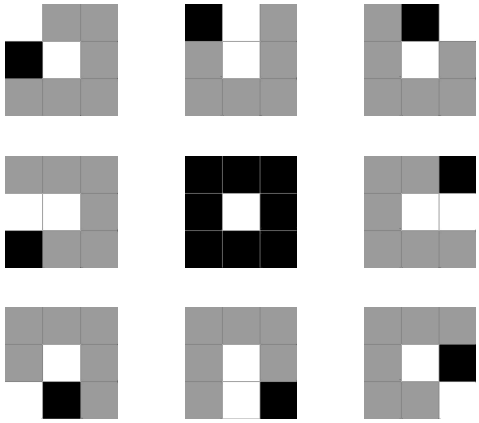


Figure 5.3: The figure at the center corresponds to an isolated pixel. The surrounding figures illustrate the different cases of degree incrementation. White and black pixels correspond to foreground/skeleton and background pixels respectively. Gray pixels are ignored in each particular case.

Then each search will discover adjacent skeleton pixels from the starting nodes and progressively mark them with the same identification number as its starting node. Each new discovered skeleton pixel is appended to the corresponding queue (the queue keeps track of the edge pixels). If a search encounters a pixel that is already marked, a path is discovered. It connects the two starting nodes having the two identification numbers to form only one edge. The figure 5.4 illustrates the edges computation.

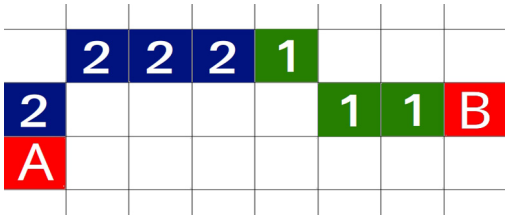


Figure 5.4: Illustration of the edge calculation. Red pixels A and B are two starting nodes. Green (resp. blue) pixels are enqueued to the search starting at the pixel adjacent to node B (resp A). Once one of the searches reaches a pixel already marked by another search, an edge between A and B is discovered.

The length of the edges is calculated as the search is performed along the pixels. Vertical and horizontal steps count as one unit while diagonal steps counts as $\sqrt{2}$ unit.

5.1.4 Graph filtering

Despite the preprocessing, once graph has been extracted, it can still be improved during a *graph filtering* procedure. Multiple graph filtering algorithms can be applied to correct

and improve the relevance of the extracted graph. Using the Python module `Networkx`[3], it is pretty straightforward to perform basic operations on the graphs. The performed improvements may rely on characteristics of the graph such as the edges size or the size of the connected components.

5.1.4.1 Filter edges size

A first and simple filter that can be applied is to remove all the edges with a length smaller than a threshold. If the removal of an edge lets a node isolated, this node is also removed. The choice of the threshold depends on the resolution of the image. Because the image resolution is normalized, a fixed threshold can be chosen.

In practice, this filter works fine for nodes and edges generated on noisy parts of the skeleton. However, it can sometimes generate a loss of connectivity in a part of the graph that was connected before. And this is often a property that one wants to preserve.

5.1.4.2 Merge and link

To remove irrelevant edges or nodes while preserving connectivity, a second idea is introduced. Is consist in replacing a set of very close nodes by a unique new node. The edges removed by the suppression of nodes should be replaced by new ones linked to the new node and their length updated accordingly. The new location should be at the mean of the coordinates of the replaced nodes. This may be weighted with the degree of the replaced nodes. The distance between the new and replaced nodes should be inversely proportional to the degree of the replaced node.

5.1.4.3 Connected component filtering

This graph filtering procedure is based on the following hypothesis : relevant parts of the graph are contained in its largest connected components. This assumes that the smaller components are results of the noisy parts of the image. Therefore, components of the graph having a insufficient number of nodes are simply removed. They often correspond to nodes and edges found in isolated characters, small shapes or drawings contained in images.

5.2 A coordinates dependent search

The first search technique that presented here is *coordinates dependent*, which means that information about the coordinates of the points is used and processed during the matching procedure. In this section, the *graph structures* used is presented before elaborating on each step of *the algorithm* and finally briefly analysing the *complexity*.

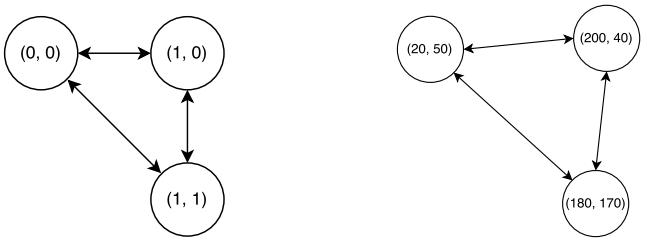
[3]See appendix A.3 for more details

For some illustration of the different steps of execution of the whole *coordinates dependent search*, refer to the Appendix C.1. For an evaluation of the search heuristic presented in this section and the resistance of the algorithm applied on noisy instances, see the section 7.2.

5.2.1 The graph structures

As previously stated, the search presented here, makes use of the coordinates of the nodes in the graph. For the purpose of this search, an *enriched directed graph data structure* is used. A common graph is composed of nodes and edges. In the enriched graph data structure used here, a *label* is added on each node and each edge. It contains additional information about the nodes and edges and, in that way, fixes the graph in the 2D plan :

- The **node-label** contains the 2D-coordinates of the node.
- The **edge-label** is composed of a *length* (always positive) and an *angle* describing the position of the neighbour[4]. The *length* is the distance along the edge between two neighbours and the *angle* is the orientation of the neighbour counted counter-clockwise from the horizontal (like in the trigonometric circle).



(a) Example of *mGraph*

(b) Example of *sGraph*

Figure 5.5: Examples of *mGraph* and *sGraph* with nodes labeled with their coordinates

The input of the *search algorithm* that will be presented after, is nothing else but a model graph *mGraph* and a search space graph *sGraph* encoded according to this nomenclature. In the figure 5.5 and table 5.1, you will find the adjacency matrices stored for an example of *mGraph* and *sGraph*. The lengths used are equivalent to a multiplicative factor close. For the *mGraph*, a convention could be to assign a unit value 1 to the shortest length. For the *sGraph*, a convention could be the length in terms of pixels between two nodes obtained during the graph extraction step.

[4]This information seems to be easily computed with the coordinates of two adjacent nodes and seems thus redundant. In fact, the length distance is not always the distance in straight line. If a curve between 2 nodes appears, the length is the length of that curve. Furthermore storing that information eases the future research.

From \ To	$(0,0)$	$(1,0)$	$(1,1)$
$(0,0)$	-	{1, 0}	$\{\sqrt{2}, 315\}$
$(1,0)$	{1, 180}	-	{1, 270}
$(1,1)$	$\{\sqrt{2}, 135\}$	{1, 90}	-

(a) Enriched adjacency table for *mGraph*

From \ To	$(20,50)$	$(200,40)$	$(180,170)$
$(20,50)$	-	{180.3, 3}	{200, 321}
$(200,40)$	{180.3, 183}	-	{131.5, 261}
$(180,170)$	{200, 141}	{131.5, 81}	-

(b) Enriched adjacency table for *sGraph*

Table 5.1: Adjacency tables of a *mGraph* example and a *sGraph* example

In the adjacency matrices, the values α and β in the tuple $\{\alpha, \beta\}$ respectively correspond to the length and the angle in degrees. Such a tuple is called an *attribute*. A node is then fully described by its *coordinates* and a *list of attributes*.

As an example, in the *sGraph* example, to reach the node $(200, 40)$ from $(20, 50)$, the attribute $\{180.3, 3\}$ indicates that a distance of 180.3 with an orientation of 3° from the horizontal must be travelled.

5.2.2 The search - Starting nodes

The different steps of the search will now be explained. The first one consists in finding what is called *starting nodes*. In the input of the search, only the two enriched graphs *mGraph* and *sGraph* are available. The first one is defined by the user and the second one is extracted from the input image. In order to start the search, a starting point of the model must be chosen. Instead of choosing a random $mNode$ and assigning to it a random $sNode$ value, it is interesting to begin the search with a judiciously chosen $mNode$.

The goal of the first step of the search is to order the possible starting nodes. By observing and analysing the nodes of the model, the most particular ones are proposed first followed by the more common ones. Those particular nodes are expected to have fewer corresponding space nodes than the average. This controlled enumeration should increase the probability of having a good starting node for the search.

To identify particular model nodes that could be good starting nodes, the set of attributes of each node are compared with each other. If two nodes have the same set of attributes, these two nodes are less particular than a node having a set of attributes appearing only once.

As an example, consider the *mGraph* defined by the user for a sudoku. It is composed of 100 nodes labelled from $(0, 0)$ for the upper left corner to $(9, 9)$ for the bottom right corner. For that model, some interesting starting nodes are the four corners $(0, 0)$, $(9, 0)$, $(0, 9)$ and $(9, 9)$ each of them having a unique set of attributes. For instance, the upper right corner $(9, 0)$ has the set of attributes $\{\{1, 180\}, \{1, 270\}\}$ to reach its two neighbours $(8, 0)$ and $(9, 1)$. The figure 5.6 shows the set of attributes for the model node $(9, 0)$ of

a sudoku.

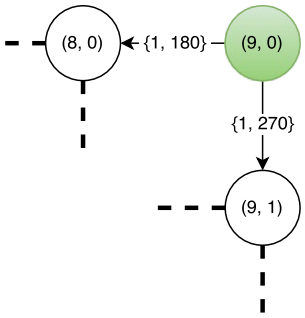


Figure 5.6: Set of attributes of the model node $(9, 0)$ of a sudoku

The procedure that iteratively returns the starting node simply scans the set of attributes of every node of the *mGraph* and then firstly returns the one(s) having a set of attributes appearing the less frequently and then the more common one(s).

5.2.3 The search - Basic matching

Once some interesting starting nodes have been detecting, some searches are initialized from the first best starting node, then from the second best one, and so on. Let's denote by SN the current best starting node. The goal will then be to try to find some *basic matchings*. When some basic matchings are found, the search can begin and progressively grow. The selection of these *basic matchings* can be for example to select the 1000 *best* ones. This selection is a *greedy-like scheme*, and permits to perform the search and a reduced search space tree. An evaluation of this *basic matching ordering heuristic* is available in the Appendix 7.2.1

The principle to find some basic matchings is the following. Compare the set of attributes of the SN with the set of attributes of each *search space node*. If two nodes have similar sets of attributes according to a tolerance on the measures and according to a *transformation law*, then a basic matching is found.

Here is an example to understand how it works. Imagine one tries to match the starting node $(9, 0)$. When comparing its set of attributes with the node $(117, 12)$ of the *sGraph* having a set of attributes $\{\{42, 187\}, \{46, 273\}\}$, a *basic matching* will be found having the *transformation law* $TL = \{ratio : 42, orientation : 7\}$ which indicates that an attribute of the model must be transformed according to the transformation law before being compared to the search space attribute[5]. Here, we have a basic matching between the model node $(9, 0)$ matched to the search space node $(117, 12)$ with a transformation law $TL = \{ratio : 42, orientation : 7\}$. The table 5.2 illustrates this.

[5]The *length* must be multiplied by the *ratio* of the transformation law and the *angle* must be added to the *orientation* of the transformation law.

	Comparison attr1		Comparison attr2	
	Value	Result	Value	Result
mNode $(9, 0)$	$\{1, 180\}$	The two attributes are perfectly aligned	$\{1, 270\}$	The length and orientation errors are tolerable (46 instead of 42 and 4°)
sNode $(117, 12)$	$\{42, 187\}$		$\{46, 273\}$	

Table 5.2: Basic matching selected with a transformation law $TL = \{ratio : 42, orientation : 7\}$

5.2.4 The search - A BFS generation of the graph - Basic scheme

Starting from a basic matching composed of a *model node* corresponding to a *search space node* according to a *transformation law*, the search itself can begin. The principle consists int progressively generating the *mGraph* and find its correspondence in the *sGraph* in a BFS way.

Theoretical explanations :

1. Given a basic matching `[mSN → sSN]`, where `mSN` is the *model starting node*, `sSN` is the *search space starting node* and a transformation law `TL`, generate the theoretical mapped set of attributes of `mSN`, denoted by $\texttt{Attr}^{TL}_{mapped}\texttt{(mSN)}$.

   $\texttt{Attr}^{TL}_{mapped}\texttt{(sSN)}$ is thus a list of theoretical attributes that should corresponds to the attributes of `sSN`, denoted by `Attr(sSN)`.

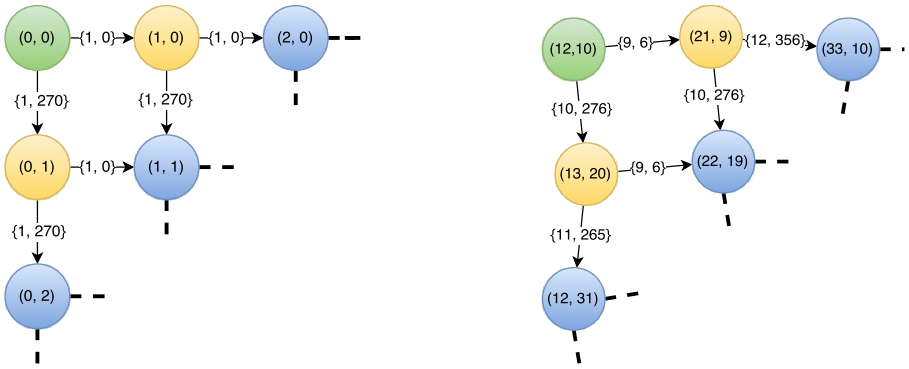
2. The second step consists then in comparing the theoretical attributes $\texttt{Attr}^{TL}_{mapped}\texttt{(mSN)}$ with the real attributes of `sSN` in the *sGraph* denoted by `Attr(sSN)`. If two attributes, one of each set, let's say $\texttt{Attr}^{TL}_{mapped}\texttt{(mSN)[i]}$ and `Attr(sSN)[j]`, correspond to each other according to a tolerance on it, then the corresponding points correspond to each other. A new pair of points is discovered/generated and the current matching is extended to become `[mSN → sSN, NB(mSN)[i] → NB(sSN)[j]]` where `NB(n)[a]` is the neighbour of the node `n` corresponding to the attribute `a`.

3. Every attributes generated in $\texttt{Attr}^{TL}_{mapped}\texttt{(mSN)}$ are compared to the attributes in `Attr(sSN)` to examine if they can possibly match. The correspondences found are added to the current matching.

4. Starting from the last correspondences found, the step 2 started again and the matching is generated in that way.

Here is a practical example of this BFS search applied to the model and search graphs shown in figure 5.7. Only the relevant attributes are drawn in the figure. The transformation law computed is `TL = {ratio:10, orientation: 6}` which perfectly aligns the

model attribute `{1, 270}` with the search attribute `{10, 276}`. Finally, the basic matching becomes `[(0, 0) → (12, 10)]`.



(a) Example of *mGraph* to match

(b) Example of *sGraph* where to find the matching

Figure 5.7: Examples of graphs where the BFS is applied. The transformation law is `TL = {ratio:10, orientation: 6}`. The green nodes forms the *basic matching*, the yellow ones are those discovered at the depth 1 of the BFS and the blue ones are those discovered at the depth 2 of the BFS.

With such examples of graphs, the BFS will progressively generate the matching as follow :

1. The basic matching is `[(0,0) → (12,10)]`.
2. According to the transformation law we have $\texttt{Attr}^{TL}_{mapped}\texttt{((0,0))} = \texttt{\{\{10, 276\} \{10, 6\}\}}$ which must be compared with `Attr(12,10)={{10, 276}, {9,6}}`.
3. We then observe that `{10, 276}` that is similar to `{10, 276}` and that `{10, 6}` is similar to `{9, 6}` according to a tolerance. The matching is then extended to `[(0,0)→ (12,10), (0,1)→ (13,20), (1,0)→ (21,9)]`.
4. The previous mechanism is then repeated for the last correspondences found, i.e `(0,1)→ (13,20), (1,0)→ (21,9)]`
5. The blue points can then be discovered.
6. Finally a matching covering all the nodes is established : `[(0,0)→ (12,10), (0,1) → (13,20), (1,0)→ (21,9)], (2,0)→ (33,10), (1,1)→ (22,19), (0,2)→ (12,31) ]`

5.2.5 The search - A BFS generation of the graph - Improvements

The basic scheme previously presented is a little bit to simple and must be improved to provide better results. In this section, some improvements are presented.

The first improvement brought to the implementation consists in making use of an *adaptive transformation law*. Instead of having a fixed transformation law that will not evolve during the search, a new adapted transformation law is assigned to each new discovered point. If the point P' has been discovered from the point P with a transformation law TL according to a tolerance, a new adapted transformation law TL' is computed and assigned to the point P' such that if the law TL' had been applied from point P, the point P' would have been perfectly aligned. Making use of such an adaptive transformation law is much more efficient that using a fixed transformation law. Indeed with a fixed transformation law, the higher the depth of the BFS is, the more some small inexactitudes are being propagated and new points will not be correctly discovered. Furthermore, updating transformation laws allows to deal with some warped or skewed graphs compared to the perfect model.

The next interesting thing to do is to explore nodes further away than in the *1-neighbourhood*[6] if no corresponding node has been found in that *1-neighbourhood*. In this perspective, nodes are incrementally looked for in the 1-neighbourhood and then in the 2-neighbourhood. This larger exploration improves the performance when some noisy points are present on some edges.

5.2.6 The search - Pseudo code and termination

To summarize all the steps of the *coordinates dependant search*, have a look at the pseudo code 5.1 which summarizes all of this.

In order to limit the search presented here, the function returning the basic matchings will, at most, returns the 500 more interesting basic matchings. The assumption is made that if none of those basic matching, quite particular, leads to a good detection of *mGraph* in the *sGraph*, then it is unlikely to uniquely and correctly realize a such detection.

Different schemes of termination are possible. A time limitation, an accepting measure of the quality of the matching (for example the number of nodes discovered), or as presented in the pseudo code here, wait that all the possible matchings, starting from the 500 more interesting basic matchings, are computed and returns the best one. Our implementation stops if 95% of the model edges have been matched for a given basic matching or stops after the 500 basic matchings have been analysed.

5.2.7 Analysis of the complexity

Remember the awful exponential complexity for an exhaustive analysis of all the possible matchings. With the numerous heuristics developed and used here, the complexity

[6]The n-neighbourhood of a node is the set of nodes being at most at a distance n in terms of number of edges.

```
input :      model graph -mGraph, search space graph -sGraph
output :     the best matching found -bestMatching
init :       -bestMatchings an empty list
function :   coordinatesDependantSearch

  startingNodes ← getStartingNodes(mGraph)
  basicMatchings ← getBasicMatchings(startingNodes) #500 returned
  for each bM in basicMatchings :
    matching ← generateMatching(bM, mGraph, sGraph)
    if matching is acceptable :
      bestMatchings.add(matching)

  bestMatchings ← bestMatchings sorted by decreasing number of
   edges matched
  return bestMatchings[0] #the best matching
```

Pseudo-code 5.1: Coordinates dependant search - pseudo-code

drastically drops down and it becomes acceptable to compute graph matchings for graphs of more than one hundred nodes. The computation of the complexity of the coordinates dependant search is available in Appendix C.3.

5.3 A coordinates independent search

In this section, a *coordinates independent search* is presented. This approach is inspired by *TALE* : a *Tool for Approximate Large Graph Matching*[7] [21]. The goal is to identify an approximation of the model in the search space graph by focusing on its *structure*. Structure refers here to nodes and links only. Neither nodes nor edges carry information relative to its location or attributes. We want to identify an *approximation* of the model graph in the space graph. Some nodes or edges may be missing or have slightly different structure.

The first step consists in defining how a *model node* can possibly match with a *search node* when only the structure of the graphs is considered. Then a two-steps procedure similar to the coordinates dependent search described before is executed : specific model nodes are identified and matched first, afterwards this basic match is incrementally extended to form a final match. Finally, from several final matches obtained with the different basic matches, one or k-best ones are retained.

[7] In the remaining of the thesis, we intercheagibly use *TALE-like approach* and *coordinates independent approach*

5.3.1 Nodes matching

If two nodes, one from the model and one from the space, are similar enough, we say that a match has occurred. Because a model node can possibly match with several search node, a *quality evaluation of a match* is also required. The node matching procedure presented here is based on the study of the neighbourhoods of the compared nodes. To evaluate the similarity between two neighbourhoods, several characteristics may be considered : the *degree of the nodes* involved, the *links between their neighbours* or their possible *labels*. Other criteria can be used and combined to specify properties that must be met for a match to occur. When facing specific problems, some characteristics are more relevant than others and should be selected accordingly.

5.3.1.1 Match condition

When comparing a model node with a search node, we want to determine if they can possibly match together. To make a decision, some *matching conditions* must be defined. To compare the nodes, a *label* is assigned to each node and comparing two nodes is equivalent to compare their respective label. A naive *label-choice* consists in using the degrees of the nodes $mNode$ and $sNode$. To perform an approximate match, a number $nb_{miss} = \rho * mNode.degree$ of missing nodes in the neighbourhood of $mNode$ is allowed. It is proportional to the total number of neighbours expected. The value ρ is user defined and sets the degree of the approximation. It corresponds to the percentage of missing neighbours allowed. A match occurs if the following condition is respected :

$$mNode.degree - nb_{miss} \leq sNode.degree \tag{5.1}$$

meaning that *sNode* has enough neighbours to correspond to *mNode*, with nb_{miss} misses allowed.

The number of nearby nodes is not enough to confirm a match. A new *label* adding the information about how those nodes are linked together improves the evaluation. The amount of edges linking the neighbour nodes together is used as a second proximity indicator. If nb_{miss} limits the number of missing neighbours allowed, the maximal number of missing connections between those neighbours can be determined. In the worst case, the nb_{miss} missing nodes are connected to each other and to the $mNode.degree$ remaining nodes :

$$nbc_{miss} = nb_{miss}\frac{(nb_{miss} - 1)}{2} + (mNode.degree - nb_{miss})nb_{miss}$$

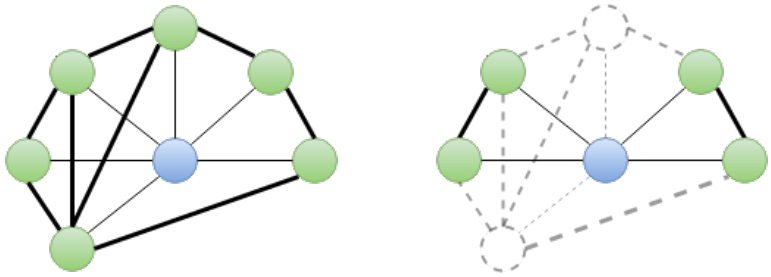
Therefore, when considering the relation between their respective neighbourhood, a match occurs if the following condition is respected :

$$mNode.nbc - nbc_{miss} \leq sNode.nbc \tag{5.2}$$

where *nbc* gives the total number of connections between the neighbours of the nodes. It corresponds to $sNode$ having a number of neighbours connected to each other sufficient

to be matched with $mNode$ with nbc_{miss} missed connections allowed.

For example, let's try to match the center nodes (blue) of the graphs shown on figure 5.8. We have $mNode.degree = 6$, $sNode.degree = 4$, $mNode.nbc = 8$ and $sNode.nbc = 2$ (represented by bold edges). With a number of misses allowed $nb_{miss} = 1$ ($\rho \geq 1/6$), no match occurs since condition 5.1 is not respected. With a number of misses allowed $nb_{miss} = 2$ ($\rho \geq 1/3$) conditions 5.1 and 5.2 are both respected, a match occurs (the number of connections missed allowed is $nbc = 9$).



(a) Model graph (b) Space graph

Figure 5.8: A match occurs between the blues nodes only if the number of misses allowed is $nb_{miss} \geq 2$ ($\rho \geq 1/3$)

5.3.1.2 Node matching quality

Given the above conditions, multiple *model nodes* will match the same *space node* and vice versa, hence the need of a *confidence computation*. Again, several metrics are conceivable and customizable according to the problem. The confidence metric should be defined accordingly with the chosen match conditions.

Based on the neighbourhood similarity, the following metric is introduced. Let $\widetilde{nb}_{miss}$ and $\widetilde{nbc}_{miss}$ be the actual number of missed neighbours and missed neighbour connections. For example, on the figure 5.8, 6 neighbours and 8 connections are expected while 4 and 2 are retrieved in the space : $\widetilde{nb}_{miss} = 2$ and $\widetilde{nbc}_{miss} = 6$. The fraction of missing neighbours and missing neighbour connections are given by $f_{nb} = \frac{\widetilde{nb}_{miss}}{mNode.degree}$ and $f_{nbc} = \frac{\widetilde{nbc}_{miss}}{mNode.nbc}$, which we want both to be minimal. Therefore, a quantification of the miss can be given by :

$$m = \begin{cases} f_{nbc}, & \text{if } \widetilde{nb}_{miss} = 0 \\ f_{nb} + \frac{f_{nbc}}{\widetilde{nb}_{miss}}, & \text{otherwise} \end{cases}$$

Since f_{nbc} is correlated with the number of missing neighbours, it is adjusted by dividing it by $\widetilde{nb}_{miss}$. We have $0 \leq m \leq 2$. In our example, we have $f_{nb} = 1/3$ and $f_{nbc} = 3/4$ giving $m = 0.71$. The quality of a match, which we want to maximize is the positive weight $w = 2 - m$.

5.3.2 Step 1 : Match specific nodes

The proximity of a *node match* having been defined, let's explain the search principle. The first step of the algorithm follows the same logic described in the coordinates dependent approach. Some nodes in the model graph are distinctive from the others. How those distinctive nodes are identified should be related to the characteristics selected to elaborate a match, but can rely on any other feature considered relevant. We call *kind*[8] a fixed set of such characteristics.

Based on the neighbourhood similarity, we first define a *kind* with two values : the *number of neighbours* and the *number of connections between the 1-neighbours*. To increase the diversity of kinds and subsequently the distinction between nodes, we extend the kind with a third value : the *number of connections between the 1 and 2-neighbours* of the nodes. For example, in the sudoku problem, five different kinds appear, identified with different colors on the Figure 5.9. The less model nodes share a same kind, the more important it is considered. Those nodes are more distinguishable from others, as the four corner nodes of the sudoku, and should be better starting nodes for the step 2 of the algorithm. Another classification of kinds can be defined according to the problem.

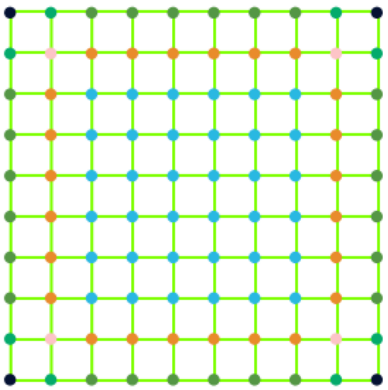


Figure 5.9: Different kinds in the sudoku model. The kind is defined by the tuple `(degree, 1-1 connections, 1-2 connections)`.

The algorithm tries to match the model nodes belonging to the most important kinds with nodes of the space. A valid match forms a *possible start couple* exploited in the second step of the algorithm.

In several applications, multiple start couples may be selected to form a basic matching. From important model nodes matched to spaces nodes, a maximal bipartite graph matching algorithm may be performed to extract a reliable set of starting couples. Then, from this basic matching, the search will expand and progressively grow the graph from multiple confident locations. A basic matching composed of multiple starting nodes

[8] We use the term *kind* which is actually equivalent to term *label* used in the literature.

represented by the black nodes on the figure 5.10.

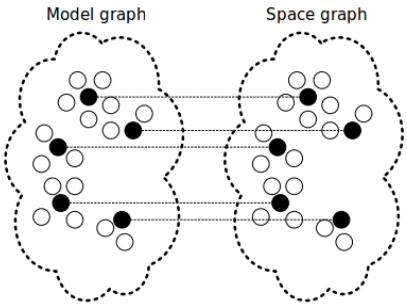


Figure 5.10: Filled black nodes correspond to the starting matched couples and empty nodes to match expanded during the step 2 of the algorithm [21]

However, this improvement must be used with caution. An inconsistent set of start couples can never lead to a consistent final graph match. For example, for the sudoku problem, the four corners are distinguishable from the other nodes and may be used to compose the model nodes of a basic matching. However, imagine that the top-left and top-right model corners of the sudoku are matched to the top-left and bottom-right corners of the search space to form the basic matching. In such a situation, the search will never lead to a correct final matching of the sudoku, despite the high quality of the start couples involved. Using such an improved scheme for the basic matching multiply in fact the combinations of possible basic matchings. It can sometimes improve the expansion step, sometimes not.

5.3.3 Step 2 : Extend the match

The second step of the algorithm consists in extending the match of important nodes obtained in step 1 to the surrounding nodes. Here, we only describe it briefly, a complete explanation and pseudo-codes is available in the Appendix B.

The step 2 starts by considering already *matched nodes couples* discovered thanks to the step 1. To extend this matching, the nodes in the 1-neighbourhood and 2-neighbourhood of the current match couple (model node and search node) are analysed. If a match occurs between a *model neighbour* and a *search space neighbour*, the corresponding nodes are added to a priority queue, along with the corresponding match quality. Once all the possible matches (in the neighbourhoods) are computed and stored in the queue, the best one is popped out and added to the final graph match. Those newly matched nodes become new *matched couples* that form the new basis to extend the current matching. Once no more new node match couple can be added to the final graph matching, the graph generation ends.

To limit the size of the queue, hence the total complexity of the search, only one occurrence of each space nodes is allowed among every node matches contained in the

queue. Therefore, when a match occurs while analysing the neighbourhoods, it replaces the match with the corresponding space node in the queue only if its quality is higher.

5.3.4 Select the best final matching

The first basic node match couple obtained at step 1 may not lead to the best matching once extended. Therefore, several *starting node matches* must be tried and the different final matchings compared in order to select the most accurate one(s). Hence the need of defining the accuracy of a final graph match. A simple choice is simply the sum of the individual quality of each matched node.

5.3.5 Summary of the available choices

Several small design decisions influence the results of the coordinate independent approach. They form the heuristic of this technique. To remain consistent, each one must be selected in accordance with the others.

- Node matching
  - What are the conditions for a match?
  - How to evaluate the quality of a match?
- Specific nodes
  - What distinguishes nodes from each others (*kinds*)?
  - How to order those kinds according to their importance?
- Extend the match
  - How to sort the priority queue?
  - Size of the lookahead (size of the neighbourhood)
  - What is a better matching for queue replacement?
- Best final match

5.3.6 Choices and improvements for sudoku like problems

As mentioned before, the heuristics and characteristics must be tuned according to the selected problem. In this section, we develop and justify our choices and improvements for a TALE-like algorithm to perform well on problems similar to the sudoku.

In the sudoku problem, there is no connection between the 1-neighbours of a node. Therefore, the node matching conditions and quality evaluation are improved by taking into account the connections between the 1 and 2-neighbourhoods. To remain consistent, the kinds selection (step 1 of the algorithm) also rely on those connections (as realized

in the *kinds* explanation).

Regarding the quality of the matches, a simple observation leads to a crucial improvement. We look at two different nodes in the space, one of small degree and one of significantly higher degree. With the basic match quality evaluation, when we try to match them with a model node having smaller degree and smaller number of neighbours connections, they both match with the same quality. The fact that one of the two space nodes have larger degree and an excess of neighbours connections does not influence the quality of the match. To overcome this, the overload is measured with the amount of excess neighbours and connections (as we evaluated the miss with the amount of missing neighbours and connections). This quantification of the overload is also subtracted from the total match quality. With this improvement, only nodes having the exact amount of neighbours and neighbour connections will have a maximal match quality.

Another problem emerges during the extension phase. When we try to select the best match in the queue, how do we deal with multiple matches having a quality equal to the best one? In our problem, the sudoku graph structure contains a lot of symmetries. For example, once the search reaches the centre of the graph, each 4-degree node is linked to four 4-degree nodes. The qualities of the matches evaluated at that point of the search are almost indistinguishable.

A first solution could be to perform a backtracking search. One could keep track of the possible matches at each iteration, arbitrarily pick one and add it to the final matching. If this decision leads to a poor result, a backtracking is realized and another match is selected at the concerned iteration. Unfortunately, for the sudoku problem it rapidly gets out of hand since a lot of symmetries can occur.

A second solution consists in analysing the state of the current matching to take the decision. Explanations : if the queue contains different potential node matching sharing the same quality, instead of randomly selecting one of those, the neighbourhood of the match couple is observed. If some neighbours are already in the matching, it is a better idea to select this potential match couple than another. This respects the following hypothesis : selecting first nodes with a neighbourhood in the current matching is a safer choice than choosing nodes with no neighbour in the current matching. The flaw of this solution is still its greediness. An early wrong decision may have a bad influence on the next node matches selection.

5.3.7 A TALE-like algorithm : Conclusion

An evaluation of this algorithm is realized in the section 7.3 on randomly generated model and space graphs. The approach described in this section is relevant to find defined model structure in graphs. Good results are characterized by subgraphs of the space having nodes with degree and numbers of neighbours connections corresponding to the nodes of the model. It performs well in application where only a structure is researched such as biological datasets or social networks.

However, it behaves less well in applications like the retrieval of a very symmetric graph, as the sudoku problem. In the latter case, the grid describing the sudoku is accurately identified in a clean space graph, where almost no additional nodes nor missing edges disturb the search space. Nevertheless, when noisy structure appears in the search space, a wrong early decision can be made when selecting the node match couple to add to the current matching. This early wrong decision rapidly leads to an undesired result when identifying model graphs that need to be exactly matched. For example, when the model graph describes a grid, even if only a small amount of nodes are not exactly matched in the space, the resulting graph identified may be far from and grid-like graph. Indeed, a single inversion of nodes corresponding to cells' corners, while being evaluated as a high quality match, will break the sought pattern of the cell. This result is a disaster for the next step which aims at detecting the data out of the cells. This is an observed result of the tests performed in the section 7.3. A relatively good model structure, relying on degree and neighbours connections, is retrieved in the space but the exact model is not.

5.4 A constraint programming search

As already stated in the beginning of this chapter, the graph matching problem faced here is nothing else but an optimisation problem. Indeed, the aim is to find the *"best"* matching between the *model nodes* and the *search nodes*. And constraint programming seems to be a very convenient tool to optimize such a problem.

In the two previous sections, two *greedy-like searches* have been presented. Some heuristics are used to choose good starting points, to choose good values to attributes to the nodes and to choose the promising next variables to assign. But a question arises : *Is it not possible to use these good heuristics and include them into a constraint programming model ?*. And because an exhaustive search isn't feasible in practice due to the huge size of the search space, *why not using a search heuristic like* large neighbourhood search *(LNS) ?*

In this section two constraint programming models are considered and reviewed using the *Operational Research in Scala* library (OscaR) [14].

5.4.1 A naive model

The first basic and naive model presented here follows the natural intuition and is pretty straightforward to implement. Nevertheless, we will see that this naive implementation isn't completely correct and that fixing the issue implies to rethink all the model which will lead to the second model.

5.4.1.1 Variables definition

Let's begin to describe this model with the definition of the variables :

- `nodesModelVars` is an array of *nModelNodes* `CPIntVar`, one for each *model node*[9]. The initial domain of each $\texttt{nodesModelVars}_i$ is the set of all search space nodes at the union with a *dummy value* -1. This value is considered as a missing node. In practice it is not this domain that is used but a restriction according to the *labels* of the nodes.

  Explanations : During a preprocessing step, the *models nodes* and the *search nodes* are analysed and a *label* is assigned to each node. A label is a 5-tuple `(n1Neigh , n2Neigh, 1to1C, 1to2C, 2to2C)` where `n1Neigh` is the number of 1-neighbours, `n2Neigh` is the number of 2-neighbours, `1to1C` is the number of edges/connections between 1-neighbours, `1to2C` is the number of connections between 1-neighbours and 2-neighbours and `2to2C` is the number of connections between 2-neighbours. Using such labels makes comparisons between *model nodes* and *search nodes* possible. A distance can be established between each *model node* compared to each *search node* and the initial domain of each *model node* can be restricted to the *search nodes* at a reasonable *label-distance*.

- `valueOcc` is an array of *nSearchNodes+1* `(Int, CPIntVar)` tuples, that will be used to count and restrict the number of occurrences of each value of the domain. In this array, a `CPIntVar` is associated to each *search space node* plus the *dummy value*. The `CPIntVar` domain of each tuple $\texttt{valueOcc}_i$ is the set $\{0, 1\}$ excepted for the *dummy value* that has the domain $[0 \rightarrow \frac{nModelNodes}{2}]$ which means that at most half of the model nodes can be missing (assigned to the dummy value).

5.4.1.2 Constraints declaration

In order to find acceptable solutions and to be able to prune the search space during the search, some constraints must be declared. Here is the description of the constraints declared :

- `gcc(nodesModelVars, valueOcc)` is a *global cardinality constraint* ensuring that every $\texttt{nodesModelVars}_i$, that isn't bound to the dummy value, has a different value than any other $\texttt{nodesModelVars}_j$. This is realized via the domain of each $\texttt{valueOcc}_i$ tuple that ensures that each search node can appear only zero or one time. The only exception is for the *dummy value*.

- $\forall (i, j) \in$ modelEdges :
  `table(nodesModelVars(i), nodesModelVars(j), allowedValues)` are table constraints restricting the allowed values for each pair of nodes forming an edge in the model. The `allowedValues` table contains every possible edge of the search graph at the

[9] 100 variables for the sudoku.

union with the possible combinations of *dummy nodes*[10] to real *search nodes*. The table 5.3 illustrates the content of this `allowedValues` table.

From value	**To value**
∀ edge ∈ sGraph	
∀ node ∈ sGraph	*dummy value*
dummy value	∀ node ∈ sGraph
dummy value	*dummy value*

Table 5.3: Content of the `allowedValues` table that constraints every edge from the *mGraph*

The intuition behind this is simple : for every edge of the model, the allowed values are : every edge of the search graph plus the fictive edges starting from the *dummy value* to a search node plus the fictive edges starting from a search node to the *dummy value*, plus the fictive edge starting from and ended at the *dummy value*.

5.4.1.3 The search

A search heuristic has been used to improve the efficiency of the matching. Instead of a basic *binaryStatic* or *binaryFirstFail* search, a custom search has been defined. The different heuristics are described below :

- **Starting Node Heuristic** : If no variable is assigned yet, the first variable is randomly selected from the top 5 most particular variables (in terms of its label).
- **Value Ordering Heuristic** : The branching strategy chosen branches on every possible value of its domain ordered from the minimum *label-distance* to the maximum *label-distance*. The most promising values are tried before the less promising ones.
- **Variables Ordering Heuristic** : When already one variable is assigned, the next variable selected to be assigned is a random neighbour of the last variable assigned, if any. If none exists, a random unbound variable is chosen.

5.4.1.4 The Optimization

A initial search allowing a large amount of missing nodes is initially launched to find a basic solution. Then an LNS optimization is started during a given amount of time (100 seconds for example) to explore most of the search tree with a good diversification. The

[10] Nodes that are assigned to the dummy value; missing nodes

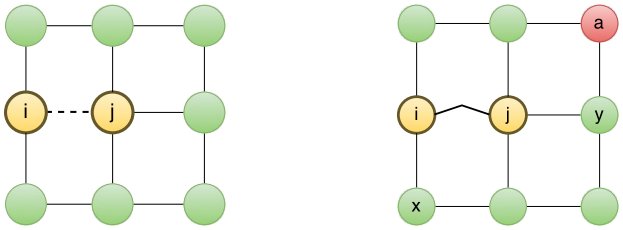
target is to minimize the `valueOcc` corresponding to the dummy value, i.e to minimize the number of missing nodes allowed.

5.4.1.5 An imperfect model

As introduced, the naive model presented here isn't completely correct. Despite the fact that this intuitive model seems to be correct, some problems lie in the table constraints.

- First the constraint only allows missing nodes and not missing edges. It correctly allows missing nodes by using the *dummy value*. But let's consider the example of a missing edge visible on the figure 5.11a. What if, in the *mGraph*, an edge is supposed to link i and j, but that this edge is missing in the *sGraph*. Well, according the table constraint, if there is an edge between i and j in the model, both i and j *can not* be assigned to a *search node* because there is no link between them. At least one of them must be assigned to the *dummy value*. Which means that the *model nodes* can only be matched to *search nodes* having all his edges correctly detected. This is an issue.

- Similarly the constraint doesn't allow noisy edges (see figure 5.11b). Because, between i and j there is no direct edge, at least i or j must be assigned to the *dummy value*.



(a) Missing edge issue

(b) Noisy edge issue

Figure 5.11: Issues with the table constraint

These limitations are very complicated to fix without redesigning the whole model :

- The *noisy edge issue* seems to be the easiest to fix. One could compute the powers of the adjacency matrix to discover the 2-paths (power 2) or the 3-paths (power 3) between the pairs of nodes and add this to the `allowedValues` of the table constraints. However, despite fixing the noisy edge issue, it also drastically drops the pruning power of the constraint since it allows every pair of nodes at a distance of 2 or 3 to be linked together. With this scheme with a 3-path tolerance, the node x would be allowed to be linked to any other node then the node a (see figure 5.11b). It could be linked to y for example.

- Fixing the *missing edge issue* is much more complicated. Indeed, the only way to fix the problem without redesigning the model is to remove the table constraint. To guide to search through a good solution, the optimization must be updated to minimize the number of missing nodes *and* missing edges. Meanwhile, all the pruning power of these table constraints flies away and the search becomes impracticable due to the huge size of the search space.

5.4.1.6 Results

This naive and imperfect model has been tested on some good instances and unfortunately poor results have been observed.

On perfect (matching a model to itself) and small (size 6) sudokus, the search is nearly exhaustive and some goods results are observed.

When increasing the problem size to a 9×9 sudoku, some runs gives the perfect matching within 100 seconds. However, due to the huge size of the search space, the partial optimization of each LNS iteration and the symmetries that are present, it often happens that the search falls into a local optimum that would requires to relax 80% of the variables to escape which is never the case with the LNS search. It can be possibly fixed by allowing some restarts. Experimentations have shown that the time limit must be considerably raised up to allow a sufficient amount of time between each restart, which is not practicable.

On real life instances with some noisy edges, missing edges, noisy nodes and missing nodes, no exploitable result has been observed. Indeed, due to the same reasons as just cited, local optimums respecting the constraints are rapidly found and very hard to escape. Furthermore, we observed that the partial solutions matching 75% of the *model nodes* and respecting all the constraints of the model are not relevant at all.

Some results are presented in the Appendix C.2.

5.4.2 A revised model

As just explained, the previous model had some issues in terms of the table constraints which could not be fixed due to the choice of the variables. A revised model, described here brings some modifications of the previous model to correct it. It is described here.

5.4.2.1 Variables definition

Instead of choosing the model nodes to be the variables, this model uses the *model edges* : `edgesModelVars`. The domain of each variable is assigned in a similar way than previously, using this time an *edge-label* comparison. The *edge-label* is the pair of the two

*node-label*s, one for each of its extremities. In a similar way than previously, an array `valueOcc` is used to limit the number of occurrences of each edge and of the *dummy edge*.

5.4.2.2 Constraints declaration

The *global cardinality constraint*, adapted for the edges, ensures that two different model nodes can not be assigned to the same search node.

This time the table constraints are established for each pair of consecutive model edges. That is, we want to ensure the connectivity at each node. These constraints enforce that if two *model edges* meet at one node in the model, than the two *search edges* assigned to the corresponding variables must meet in a node in the search graph.

With this constraints declaration, the main *missing edge issue* is fixed since a missing edge can be assigned to the *dummy edge value* with no consequence on other edges assignments. The other issue, regarding the noisy edge, is not fixed. However, according to the drawbacks in terms of pruning it involves, a such solution has not been explored.

5.4.2.3 The search and the optimization

Adapted to meet the new choice of variables, the principle of the search and optimization remains the same.

5.4.2.4 Results

This model, despite being more conform to what we want, did not provide much more significant improvements versus the naive approach. The size of the problem and the symmetries present made the search falling into local optimums which are difficult to avoid.

Again, a discussion of the results is available in the Appendix C.2.

5.4.3 Summary of the CP approaches

Constraint Programming is the golden tool to solve a bench of different optimization problems like vehicle routing problem, scheduling, nurse rostering, and so on.

Unfortunately, for the *approximate subgraph matching* problem we face here, the constraint programming approach is not advised. Indeed, due to the difficulty to establish a good optimization model, the constraints do not give a good pruning and the model only allows very approximative solutions. That's why the more greedy and guided searches approaches previously presented are more promising.

Chapter 6

Optical recognition and resolution

In the previous chapter, the *problem detection* step of the algorithm allowed to localize the problem. The two straightforward but important last steps of the algorithm are the *optical character recognition* (OCR) and the *problem resolution*. These steps respectively allow to obtain a mathematical formalization of the problem and to solve it.

These two last steps of the algorithm flow are highlighted in the figure 6.1.

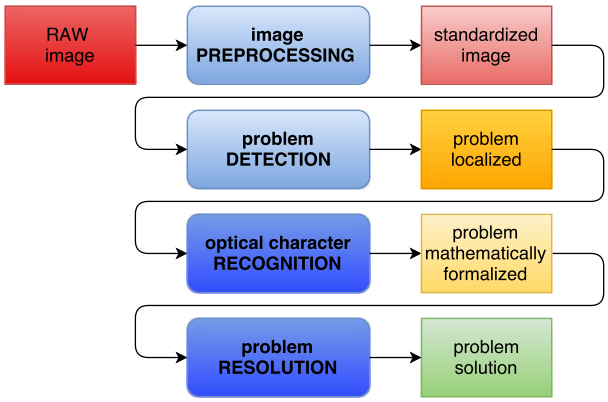


Figure 6.1: Execution flow of the algorithm - Recognition and Resolution

While the *problem detection* step can be implemented in a general way with a description of the desired problem in the input, for the two last steps of the algorithm, it is much more difficult. That's why, in this chapter, the recognition and resolution steps are presented for the sudoku problem. Nevertheless, the logic of the problem dependant implementation proposed here can be adapted for other problems similar to the sudoku.

After the *problem detection* step, the problem is localized by a *matching*. This matching, typically a dictionary structure, maps every model node to a search space node.

Let's determine a formalism for the sudoku problem. Let's name C_{ij} the cell of the sudoku at the intersection of the ith column (X-axis) and the jth raw (Y-axis)[1]. C_{11} and C_{99} are then the top left cell and the bottom right cell of the sudoku respectively. Let's also say that the model graph has been defined so that each node has an ID (k, l). This ID is such that the cell C_{ij} is surrounded by the four nodes $(i-1, j-1)$, $(i, j-1)$, (i, j), $(i-1, j)$ being the four corners of the cell starting from the top left corner in a clockwise way.

Using this nomenclature, the figure 6.2 illustrates a model graph example for a 3×3 sudoku.

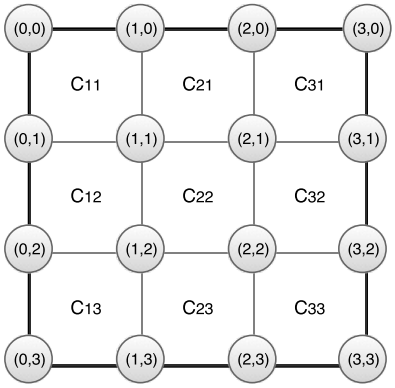


Figure 6.2: Graph model example for a 3×3 sudoku

The matching, which has become available thanks to the problem detection step, maps each model ID to a search space ID. The search space ID is in fact the coordinates of the search node in the input image where the problem lies. An example of a typical matching for a 2×2 sudoku can be seen on the table 6.1.

Model ID	(0,0)	(1,0)	(2,0)
Search ID	(20,30)	(31,29)	(42,31)
Model ID	(0,1)	(1,1)	(2,1)
Search ID	(19,41)	(33,40)	(39,42)
Model ID	(0,2)	(1,2)	(2,2)
Search ID	(21,51)	(28,52)	(38,49)

Table 6.1: Matching example for a 2×2 sudoku

The matching permits to make the *optical character recognition* to extract the data

[1]The column numbering grows from left to right and the raw numbering grows from top to bottom.

and obtain a mathematical formalization of the problem that will be used to perform the resolution of the problem.

6.1 Optical Character Recognition

The matching gives the coordinates of each model IDs which means that every sudoku cell can be localized. Meanwhile, the four corners of each cell are rarely a perfect square. It can be a rectangle, a diamond, a trapeze or a parallelogram for example. That's why the first step of the recognition is to remap each cell correctly.

6.1.1 Remapping

Using the same scheme as presented in the section 4.3.2, each cell is first remapped to a perfect square. The figure 6.3 illustrates the before and the after of a cell remapping using its four corners coordinates.

(a) Before (b) After

Figure 6.3: Remapping example of a sudoku cell

6.1.2 OCR

Once each cell is correctly remapped into a perfect square, it is then possible to proceed to the *optical character recognition*. A basic scheme has been used : the *K Nearest Neighbours*. This machine learning technique consists in teaching the computer how to classify new unseen cells knowing a set of correctly classified cells. Then when a new instance is shown, a comparison of the pixels is performed to determine which previously seen examples are the closest to the new unseen example.

Manually, some cells have been extracted, remapped to a 20×20 resolution and labelled from 0 (the empty cell) to 9 according to the number it contains. The results have been saved and are used to train a model that recognizes new unseen cells instances. A confidence of the recognition can be used to have an idea of the recognition quality. Because the aim of the OCR step is to detect the cell data for the resolution, it is important that no mistake is made during the character recognition. To this end, if the recognition attributes a similar confidence of 42% to be a 5 and of 40% to be a 6 for the same cell, then instead of attributing the unique value 5 to that cell, a domain containing $\{5, 6\}$ is attributed.

6.2 Resolution

When the problem is mathematically formalized, the resolution of a sudoku is not complicated. Using constraint programming the problem is easily solved. Let's formalize the *constraint satisfaction problem* (CSP) model for the sudoku :

- *Variables* : for each of the 81 cell, a `CPIntVar` CP_{ij} is assigned
- $\forall i, j$: $\text{Dom}(CP_{ij}) = [1 \rightarrow 9]$ determines the domain of each variables
- For each non empty extracted cells ($C_{ij} \in NE$), add the constraints $\forall\ C_{ij} \in NE : CP_{ij} = \text{val}(C_{ij})$ where valC_{ij} is the extracted value or extracted domain resulting of the OCR of the cell C_{ij}.
- To ensure the pruning and respect the constraints of a sudoku, 27 *allDifferent* constraints are added. 9 for each line, 9 for each column and 9 for each 3×3 subsquare.

A basic *backtracking depth first search* alternating *constraints propagations* and *value attributions* solve this basic CSP efficiently.

Chapter 7

Experimentations and evaluation

7.1 Problem extraction

7.1.1 Preprocessing

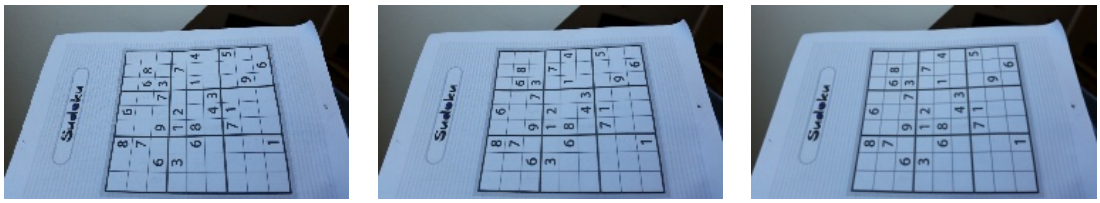
Preprocessing the image is a crucial step of our application. A poor preprocessing will give an imperfect standardized image. This can bias the problem detection and therefore result in a failure of our resolution. This section evaluates the available choices of preprocessing in order to select the most accurate one. This evaluation is not exhaustive and can be tuned according to the set of input images given to our application. The goal of the following analysis is to select parameters for "average" images.

7.1.1.1 Resizing

Three interesting mapping procedures for resizing have been explained in chapter 3 : The *nearest-neighbour interpolation*, the *bilinear interpolation* and the *area interpolation*. The goal of this section is to define the best one for our application.

We compare the result of those three techniques when realizing an important resizing. Figure 7.1 shows an image with initial pixel resolution of 3000×5000 ($15Mpx$) resized to $0.1Mpx$ images with the different interpolation schemes, preserving the width/height ratio.

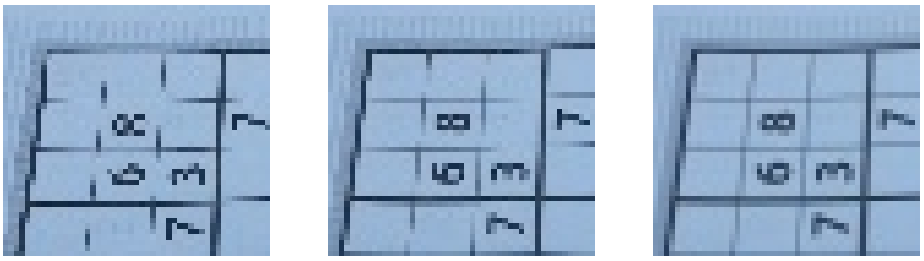
In our application, preserving the edge's information is critical, otherwise the graph detection will perform poorly. Therefore, the *area interpolation* as mapping procedure for resizing images is clearly a better choice than *nearest-neighbour* or *bilinear interpolation*. Even after performing a really important resizing, the edges are still quite apparent when using the *area interpolation*. When applying a resizing with the *nearest-neighbour* or *bilinear interpolation*, the thin black edges are flooded by the surrounding white of the image. This is a really undesired result that must be avoided. For these reasons, the *area interpolation* has been chosen for the mapping procedure.



(a) Nearest-neighbour interpolation

(b) Bilinear interpolation

(c) Area interpolation



(d) Nearest-neighbour interpolation zoomed in

(e) Bilinear interpolation zoomed in

(f) Area interpolation zoomed in

Figure 7.1: Important resizing effects with different mapping procedures

However, this difference is only noticeable for really strong resizing (here the total number of pixels is divided by 150). The average smartphone's image quality is around 10-15Mpx. For our application, we arbitrarily selected a resizing to $1Mpx$ images, while preserving the width/height ratio. This allows further processing techniques to be faster and normalized for a unique size.

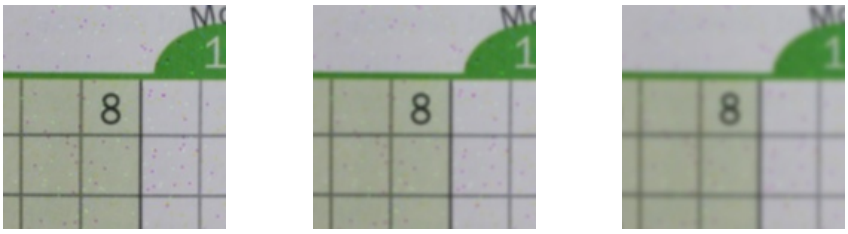
7.1.2 Smoothing

The goal of smoothing is to remove noise while preserving the important features of an image. Our analysis focuses on the use of three smoothing algorithms : *Gaussian blur*, *fast non-local mean* (FNLM) denoising, and *bilateral filtering*. The following explanations focus on the effects of the different parameters of each algorithm. They are compared in combination with the next preprocessing step : *segmentation*. It is in the segmentation step that FNLM denoising will be used in addition to the two other algorithms.

7.1.2.1 Gaussian Blur

A Gaussian blur applied to an image will merge a pixel with its neighbourhood in order to reduce the noise (pixel value that differs a lot from the neighbourhood will be smoothed). Results of different kernel size applied on a slightly salt and pepper noised image are shown in figure 7.2. A small kernel size does not remove enough noise while a large

kernel is efficient but may generate a blurry image that contains less information. Given the poor results, the blurring scheme is not used in practice.



(a) Kernel size = 3 (b) Kernel size = 5 (c) Kernel size = 9

Figure 7.2: Gaussian blur kernel size effects

7.1.2.2 Bilateral filter

Bilateral filter applied to a noisy image brings a huge improvement compared with the Gaussian blur : the use of the intensity values. Two parameters have to be tuned : σ_r (associated with the color space) and σ_s (associated with the coordinates). A larger value for σ_r means that colors highly different within the pixels neighbourhood (defined by σ_s) will be mixed together. The results of different value for σ_r and σ_s are shown in figure 7.3.

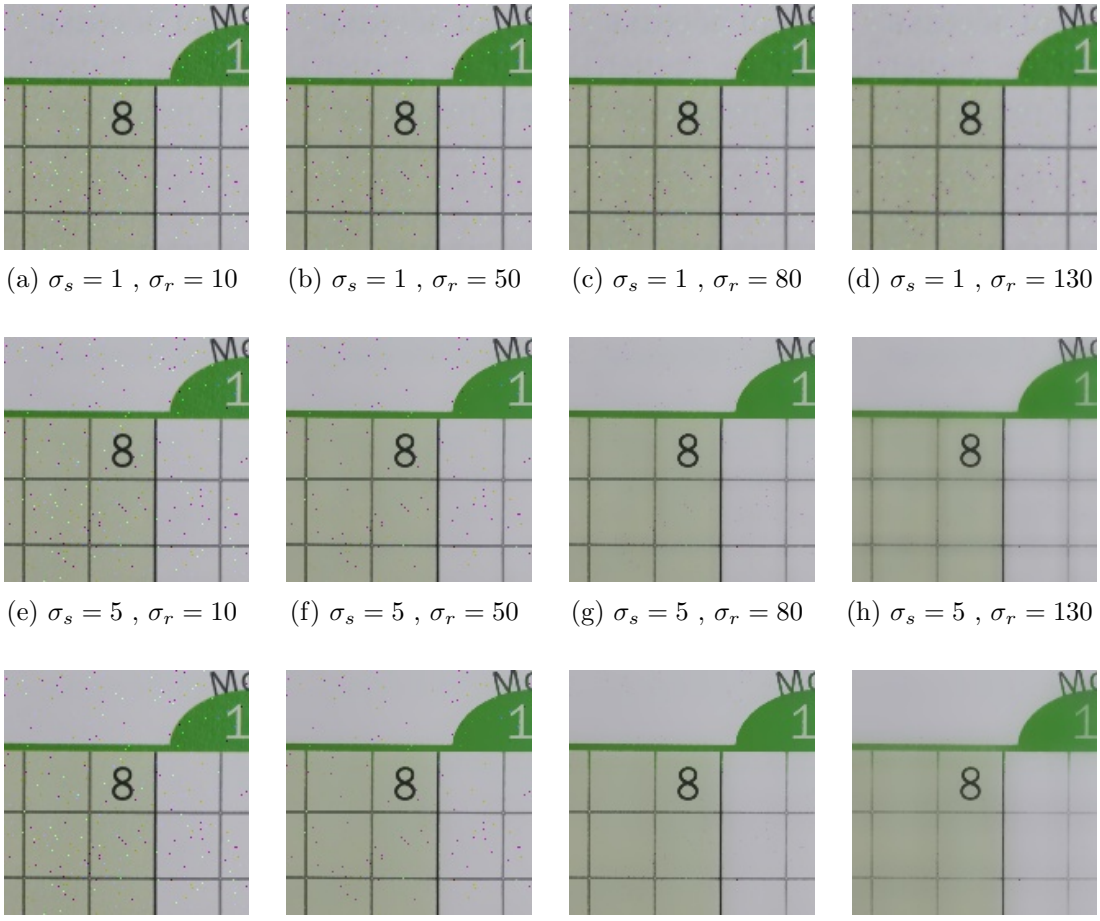
As expected, as the σ_r increases, pixels with color more distinct than their neighbourhood are smoothed as well. Like a large kernel for the Gaussian blur, here a large value for the sigma in the color space generates a too strong smoothing and information may be lost.

As the σ_s parameters increase, further pixels with close enough colors values will influence the smoothing. When a large σ_r is selected with a large σ_s, details are smoothed by the influence of the neighbourhood.

On an average image, for the bilateral filtering to be effective, the σ_r must be at least around 80 while the σ_s should be ≥ 5. In the OpenCV specification [17], it is specified that large bilateral filter sizes are slow. So our choice would be $\sigma_s = 5$ and $\sigma_r = 80$ that realizes a good smoothing while preserving edge information.

7.1.3 Segmentation

There is no doubt than an *adaptive threshold* has to be used. Using this scheme, each pixel is thresholded according to its neighbourhood. The different parameters are the *weighting policy*, commonly `MEAN` or `GAUSSIAN`, and the *size of the neighbourhood*. After the segmentation, a noise removing process, like fast non-local mean denoising, can be performed to get rid of isolated inconsistencies. However, this has to be carefully quantified in order to avoid the removal of consistent information.



(a) $\sigma_s = 1$, $\sigma_r = 10$ (b) $\sigma_s = 1$, $\sigma_r = 50$ (c) $\sigma_s = 1$, $\sigma_r = 80$ (d) $\sigma_s = 1$, $\sigma_r = 130$

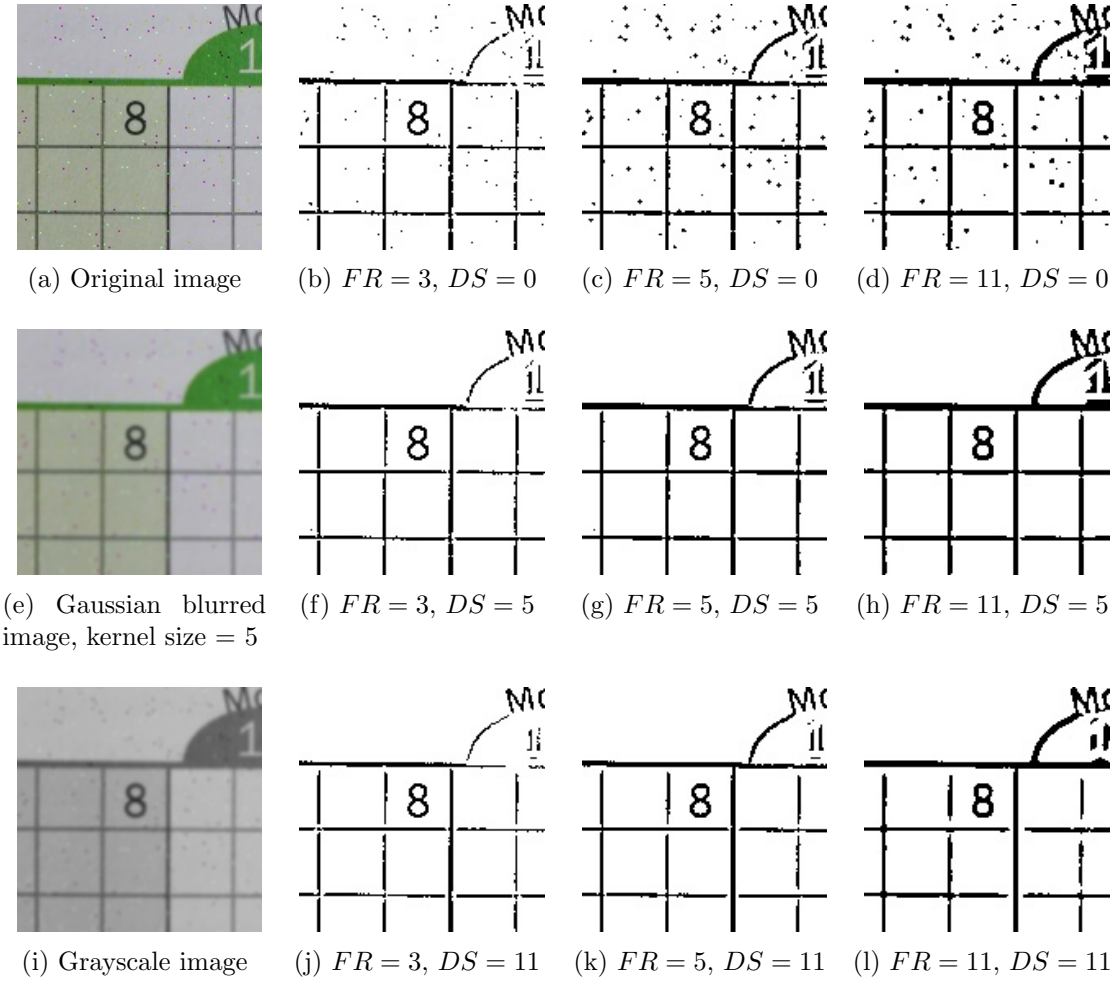
(e) $\sigma_s = 5$, $\sigma_r = 10$ (f) $\sigma_s = 5$, $\sigma_r = 50$ (g) $\sigma_s = 5$, $\sigma_r = 80$ (h) $\sigma_s = 5$, $\sigma_r = 130$

(i) $\sigma_s = 10$, $\sigma_r = 10$ (j) $\sigma_s = 10$, $\sigma_r = 50$ (k) $\sigma_s = 10$, $\sigma_r = 80$ (l) $\sigma_s = 10$, $\sigma_r = 130$

Figure 7.3: Bilateral filter space parameter σ_s and range parameter σ_r effects

7.1.3.1 Gaussian adaptive thresholding

Figure 7.4 shows the effects of different neighbourhood size choices of a Gaussian adaptive thresholding combined with two FNLM denoising strengths on a grayscale Gaussian blurred image. A small frame size does not get enough information about a pixel's neighbourhood to properly get the foreground. A large frame size gives too much weight to the center of the neighbourhood hence some noisy isolated parts are thresholded. The FNLM denoising gets rid of that effect as observed on the bottom images of the figure. But, as expected, an excessive denoising strength removes some crucial information, like some edge connectivity.



(a) Original image (b) $FR = 3$, $DS = 0$ (c) $FR = 5$, $DS = 0$ (d) $FR = 11$, $DS = 0$

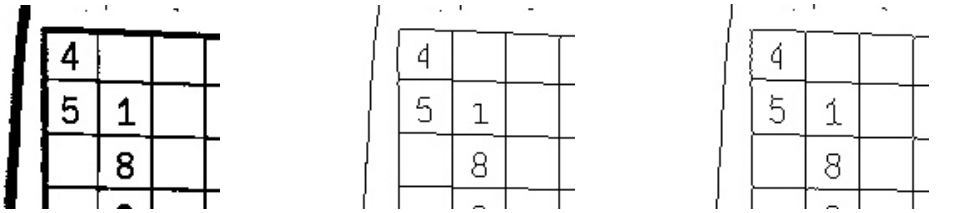
(e) Gaussian blurred image, kernel size = 5 (f) $FR = 3$, $DS = 5$ (g) $FR = 5$, $DS = 5$ (h) $FR = 11$, $DS = 5$

(i) Grayscale image (j) $FR = 3$, $DS = 11$ (k) $FR = 5$, $DS = 11$ (l) $FR = 11$, $DS = 11$

Figure 7.4: Gaussian blur and Gaussian adaptive thresholding effects combined with noise reduction. FR stands for *frame size* and DS for *denoising strength*

7.1.4 Thinning

Two thinning algorithms are presented in the section 3.5 : the *Zhang-Suen thinning* and the *Guo-Hall thinning*. They both give exploitable results for the graph extraction step of our procedure. The minor differences can establish which algorithm should be used for specific problems. Figure 7.5 shows the thinning realized by the two algorithms on a thresholded image containing a sudoku. For clarity, the *skeleton* corresponds to the black pixels and the *background* to the white ones. Zhuang-Suen thinning gives slightly straighter results than Guo-hall. However, Guo-Hall tends to preserve outer parts of the patterns. This can be noticed on the thinning of the digit 1 on the figure 7.5. Since our application focuses on extracting a graph from an image, we used Zhang-Suen algorithm to thin patterns.



(a) Thresholded (b) Zhang-Suen thinning (c) Guo-hall thinning

Figure 7.5: Comparisons of the thinning algorithms

7.2 Coordinates dependent search evaluation

In this section, we make some measurements and evaluation regarding the *coordinates dependent approach*.

7.2.1 Impact of the basic matching ordering heuristic on the search

Remember how the *basic matchings* are computed and presumed to be good starting points where to begin the search. The principle is to create a list of basic matchings. This list is created by trying to match the most particular model nodes (having the most particular feature) to some search nodes. This list has a limited size, for example 500, and is also sorted according to three criteria : *mPerc*, *sPerc* and *err* where *mPerc* is the percentage of model edges matched with the basic matching, *sPerc* is the percentage of search edges matched with the basic matching and *err* is the cumulative error of the basic matching according to the tolerance allowed. As an example, have a look at the figure 7.6.

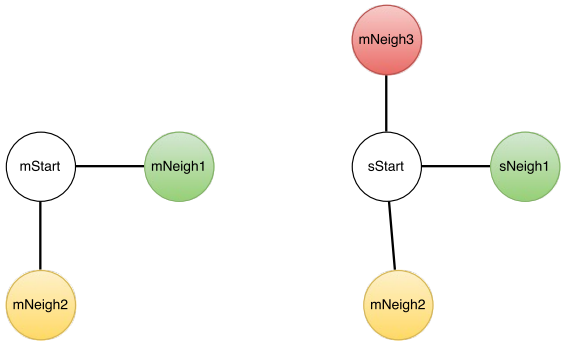


Figure 7.6: Example of basic matching

On the right, a model node *mStart* and its neighbours, on the right a search space node *sStart* and its neighbours. Consider a basic matching that matches *mStart* to *sStart*. Consider also a transformation law that perfectly aligns *mNeigh1* with *sNeigh1* (in green). Given this transformation law, we have *mNeigh2* that matches with *sNeigh2*

(in yellow) but with a little error *err*. Finally the last neighbour of *sStart*, *mNeigh3* has no match (in red). Then the score of this basic matching is given by $\left(\frac{2}{2}, \frac{2}{3}, err\right)$ since all the *model neighbours/edges* are matched, 2 out of 3 *search neighbours* are matched and the cumulative error is *err*.

As explained in section 5.2, the search tries to perform a graph generation from some basic matchings and only retains the best one. Here, the influence of the *basic matching ordering heuristic* is evaluated. On different instances, the search has been performed with the *ordering* and *without the ordering* (random order)[1]. For each test, the iteration where the best matching has been found is retained. The test has been processed on a limit of 500 basic matchings.

The table 7.1 and the figure 7.7 illustrate the comparison of the results.

Instance name	**Iteration (ordered case)**	**Iteration (not ordered case)**	**Instance name**	**Iteration (ordered case)**	**Iteration (not ordered case)**
simple1	1	282	simple2	3	384
simple3	1	92	simple4	1	67
simple5	3	96	simple6	5	129
simple7	1	63	simple8	1	147
sudoku2	5	283	curved1	3	12
curved2	14	31	curved3	37	247
curved4	3	42	dark1	36	379
dark2	3	115	default1	1	120
default2	1	73	far1	1	123
orientation1	1	148	orientation2	1	57
reversed1	8	327			

Table 7.1: Table comparison of the iteration where the best matching was found *with* and *without* the ordering heuristic on a selection of 500 basics matchings

As can be observed in table 7.1, the *ordering heuristic* gives significant results on the instances. For nearly the half (10 on 21) instances the best matching has been found at the very first iteration. And for more than 85% (18 on 21) of the instances, the best solution is found within 10 iterations.

Compared to the *not ordered case*, the ordering heuristic always outperforms it. Indeed, the iteration where the best matching is found is completely random. The best matching can be found early or very late (at the 384th iteration on 500 for the simple2

[1]Furthermore, the termination condition which stops the search if 95% of the model edges was found is disabled.

instance). The figure 7.7 illustrates the randomness when not ordered and the regularity when the ordering heuristic is used.

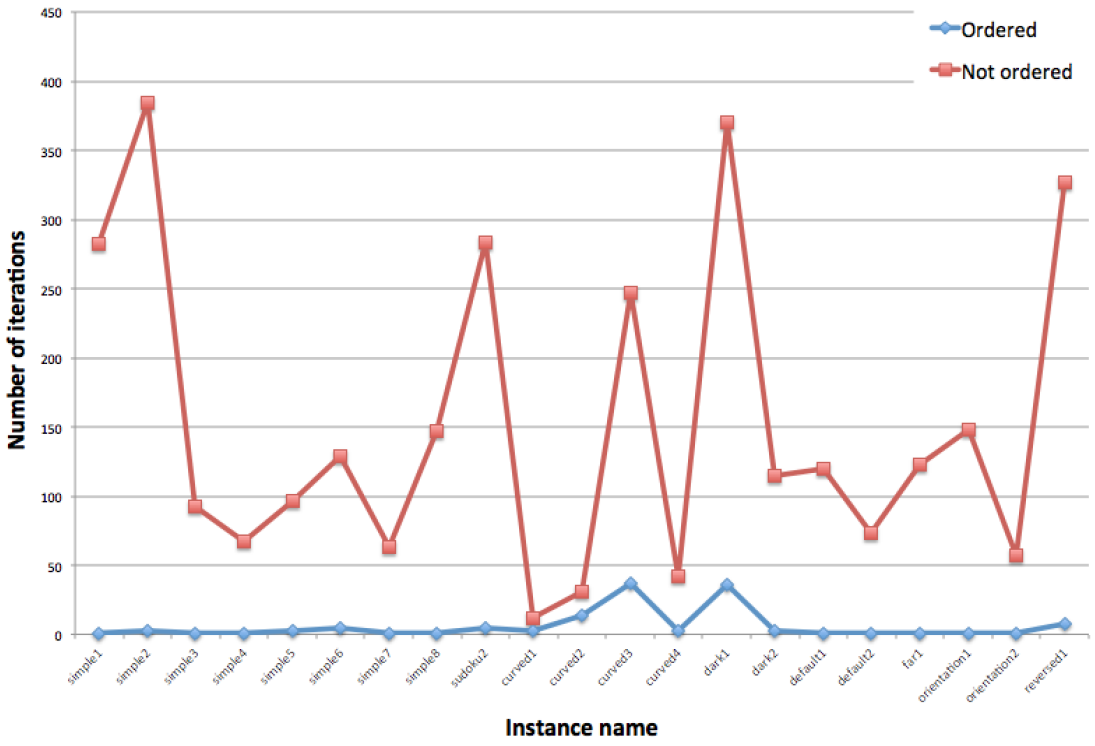


Figure 7.7: Graph comparison of the iteration where the best matching was found *with* and *without* the ordering heuristic on a selection of 500 basics matchings

Remember the termination condition of the search. It either stops when 95% of the model edges are matched in a certain matching or after the completion of the 500 iterations. Applied on good instances, the best matching is found at early iterations and then, if more than 95% of the model edges are detected, the search stops early. Evaluated on 24 different sample images, the average execution time is 2.5s per instance[2].

[2]Test executed on an *Intel i5 2.6GHz CPU* with *8Go of DDR3 ram.*

7.2.2 Resistance of the search to noise perturbations

In this section, we evaluate the behaviour of the *coordinates dependent* search on some noisy instances, to see how it reacts.

The figure 7.8 illustrates the matching obtained by the search on a noisy instance. On the left the *model graph* is represented and on the right the *search space image* is represented. The model is a 9×9 sudoku and the search space image represents a sudoku where a corner have been torn up. The matching is represented using a color code. On the model

- the small red dots represent the model nodes that haven't been matched.
- the medium coloured dots represent the model nodes for which a search node match was found.
- the big coloured dot represents the starting node of the basic matching.

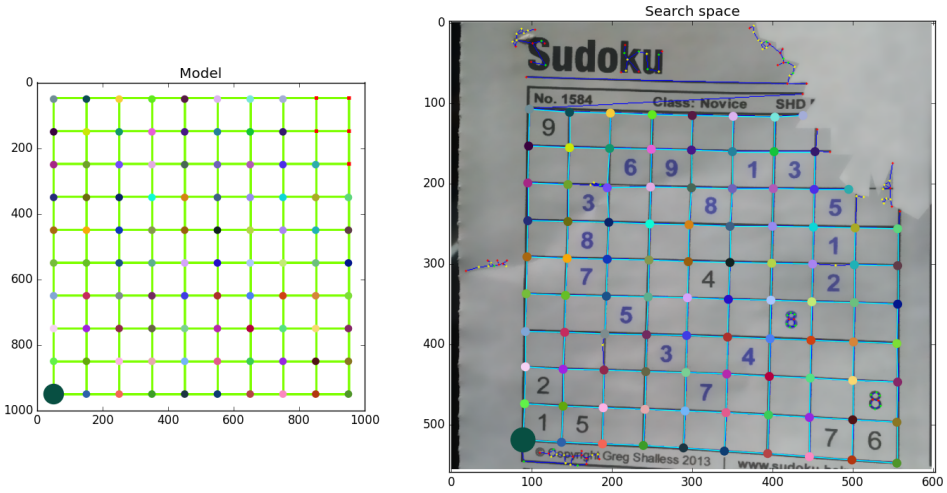


Figure 7.8: Matching obtained on a noisy instance by the coordinates dependent search.

On the search space,

- the medium and the big coloured dots are the search spaces nodes corresponding to the same coloured model nodes.
- the small dots represent the remaining search space nodes that haven't been matched.
- the dark blue lines are the search edges unmatched .
- the cyan lines are the search edges matched.

As can be observed, the search behaved well and only the correct model nodes have been matched. Even with a part of the sudoku undetected, the data can be extracted and the sudoku can still be solved.

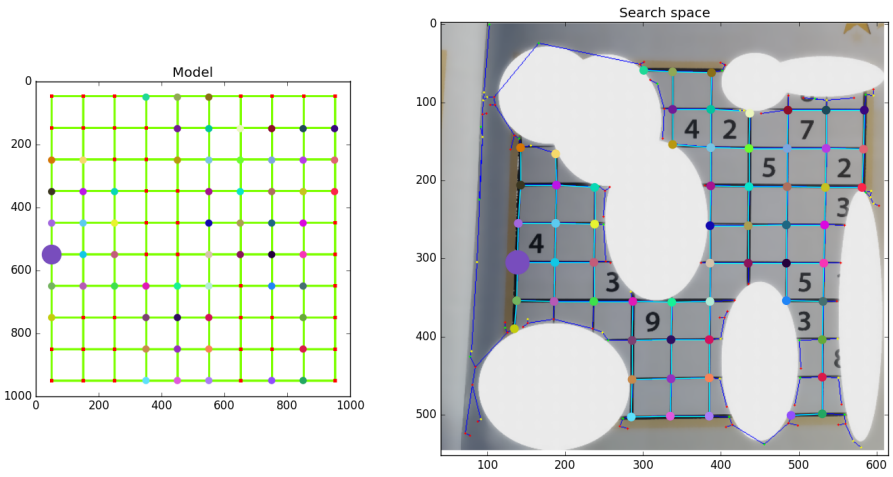


Figure 7.9: Matching obtained on another noisy instance by the coordinates dependent search.

The figure 7.9 illustrates another example where more than half of the nodes are missing. As can be observed, despite being very noisy, the coordinates dependent search manage to find every correct model node, the other being unmatched. This time, the starting node is no longer a corner since none of them is present in the search image. Instead, a node of the side of the sudoku, a little less particular than a corner, has been selected for the basic matching.

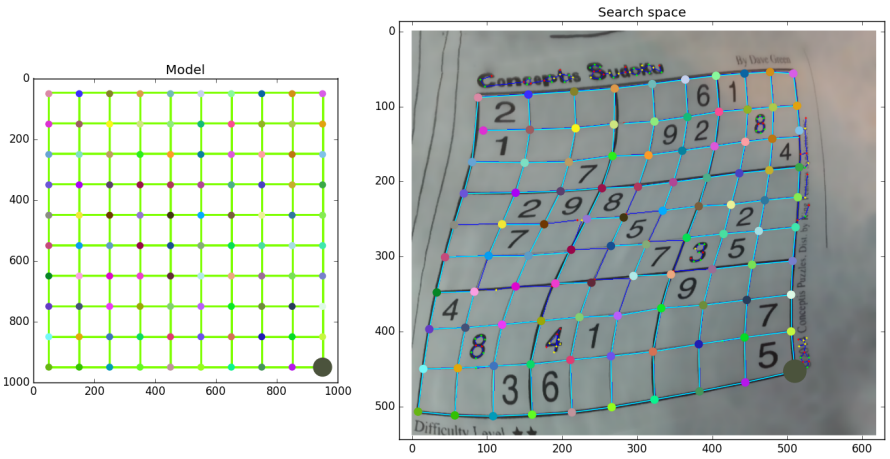


Figure 7.10: Matching obtained on warped instance by the coordinates dependent search.

As explained, an great improvement of the *coordinates dependent search* has been to update the *transformation law* at each iteration of the search. The figure 7.10 illustrates how this improvement allows to find graphs that are very distorted with a careful update of the *transformation law* at each iteration.

Finally, an instance combining a missing part of the image and being warped has been tested. As can be observed in figure 7.11, the search algorithm gets through quite easily and all the relevant nodes of the model graph has been detected.

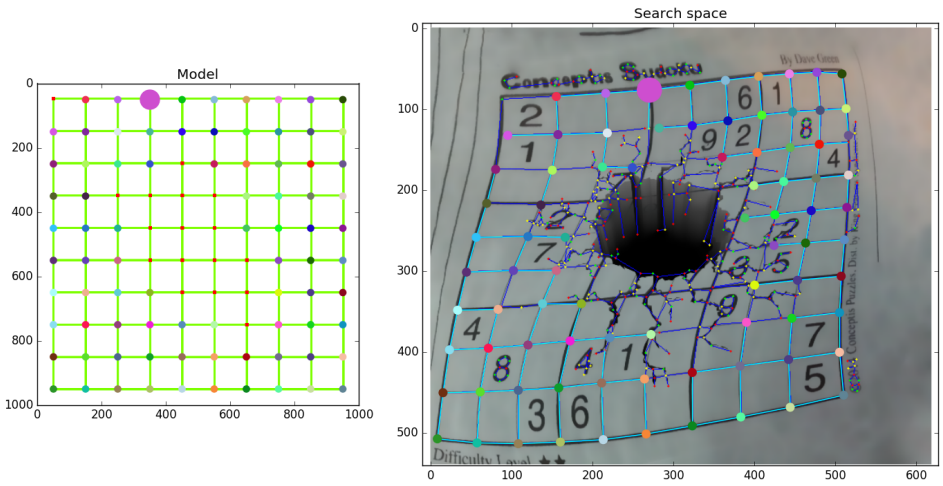


Figure 7.11: Matching obtained on warped and noisy instance by the coordinates dependent search.

7.3 TALE-like approach

In this section, we evaluate the performance of our *TALE-like approach* to retrieve a model structure in a search space. We will see that the retrieved graph may not always correspond to the expected model contained in the space graph when a lot of symmetries are present in the model. This is how we proceed :

First, we generate random connected graphs using the *Erdős–Rényi model* (thanks to the `gnp_random_graph` function of *Networkx* python module). Those graphs form a set of models. Then, quantified random perturbations are applied on the models to form a set of perturbed space graphs. Those perturbations follow the imperfections observed during the graph extraction : edges missing and additional irrelevant nodes on edges. Our analysis consists in evaluating how our TALE-like algorithm (described in section 5.3) performs to retrieve the models graph in the corresponding perturbed space graphs.

7.3.1 Edges missing

The tests are realized on 20 random connected graphs of approximately 100 nodes and 200 edges that will form the model graphs. The perturbed graphs are created by randomly removing 5, 10, 30 and 50 edges from those model graphs. Figure 7.12 shows the evolution of the ratio of correctly matched nodes and edges realized by our algorithm while the intensity of the perturbation increases. The left plots show the ratio of *exact matches* : nodes or edges in the space that are matched with their original correspondence in the models, before the perturbations. The right plots shows *degree matches* : a degree match occurs when a model node *mNode* of the matching is matched to a *search node* that has a degree higher or equal to degree of *mNode*. Those two latter plots illustrate how well the structure of the model is retrieved in the space despite the fact that the exact nodes or edges are not.

The different curves on each plot correspond to different degree of approximation, quantified by the ρ parameter. The red ones, $\rho = 0$, mean that no approximation is tolerated. The degree and number of connections must be exactly equal for a match to occurs. A large degree of approximation like $\rho = 0.5$ means a match occurs even if half of the degree or numbers of connections is missing.

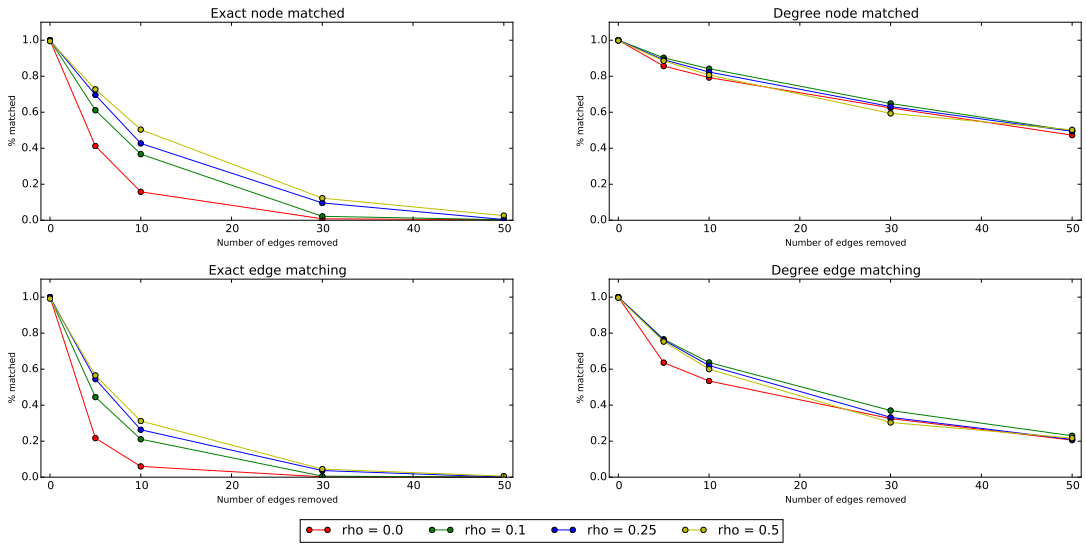


Figure 7.12: Evolution of the ratio of correctly matched nodes and edges by our algorithm while removing edges from the model graphs

7.3.2 Edges split

The same measures are realized on a second kind of perturbation. 5, 10, 30 and 50 edges are randomly selected and split by adding a new node at the centre. The figure 7.13

illustrates the ratio of *exact matches* and *degree matches* of nodes and edges as realized above.

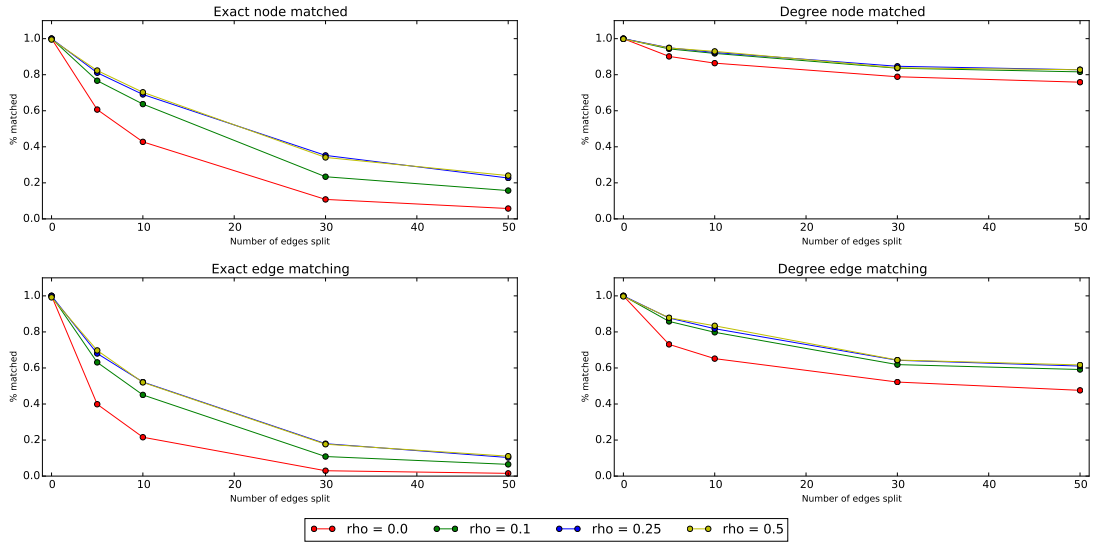


Figure 7.13: Evolution of the ratio of correctly matched nodes and edges by our algorithm while splitting edges from the model graphs

7.3.3 Interpretation

We first look at the left plots of the figures 7.12 and 7.13, corresponding to the *exact match* ratios. As the perturbation increases, the number of nodes or edges exactly matched sharply decreases. This shows the strong impact of missing edges and noisy additional nodes in the space graph. In the case of edges removal, almost no nodes nor edges are exactly matched when the number of edge removed get close to 50 (25% of the total number of edges). This is slightly better in the case of edges split perturbations, while remaining inaccurate. However, increasing the degree of approximation improves the exact match ratio. A greater tolerated approximation allows the search to realize an imperfect match that may later lead to an exact match. Those results illustrate why a TALE-like approach does not suit well for the retrieval of problems that have to be perfectly recognized to provide results exploitable for a resolution for example, like it is the case for a sudoku.

The right plots show how accurately the algorithm retrieves the structure of the model graph. In the case of edges splitting, the results are really satisfactory. Around 80% of the nodes degree can fit to the model. When the allowed approximation is sufficient, more than 60% of the model edges have their *endnodes* matched to search nodes with a sufficient degree. In the case of edges removal, a first look at the plots

of figure 7.12 may persuade the reader that the algorithm has performed poorly. The results need to be clarified. Edges are removed from the model to form the space graph. On one hand, edge removal may disconnect the graph. If it is the case, the matching found will only be part of one of the existing connected components. Large parts of the model graph can then rapidly be missing. On the other hand, even when only one edge is removed, a 100% subgraph match is not reachable any more. For example, the top-left plot shows the ratio of model nodes being match to equal of higher degree space nodes. When 25% of the edges are removed, between 25% and 50% of the nodes will have their degree reduced by the perturbation. In this case, a 50% degree node match is actually an acceptable result. The same logic may be applied to argue why the result of the degree edge matching are actually honourable.

Chapter 8

Conclusion

In this thesis, we presented several ways to overcome the common but complex challenge of identifying and solving a problem contained in a picture. A first procedure describes a resolution strongly bounded to the characteristics of a given problem. This dedicated approach makes strong assumptions about the problem's representation to be able to retrieve it. Therefore, this lack of reusability led the second and main part of our study towards an undeniably powerful data structure in computer sciences : *graphs*.

Graphs can simply yet accurately describe problems that may be contained in pictures. A procedure able to extract graphs structures from images needed to be established. This is the purpose fulfilled by the pipeline scheme described in the first section of the fifth chapter. Thanks to image processing techniques, it efficiently discerns relevant patterns and is able to build an accurate corresponding graph representation.

Subsequently, our study has been narrowed to the inspection of twos graphs : one describing perfectly the sought problem and an imperfect one extracted from the image. The objective has been to build an efficient procedure able to retrieve an occurrence of the first graph into the second one.

Since the graphs represent real two-dimensional structures, focusing on their spatial arrangements is a legitimate decision. Such a procedure, that we named *coordinates dependant*, is able to establish a correspondence by retrieving a transformation of the model in the image graph. Thanks to the spatial knowledge, we showed that this technique is really robust to perturbations that may appear in the image, such as torsion or noisy irrelevant elements.

Last part of our study focused on two *coordinates independent* procedures. Those procedures aim at realizing an equivalent retrieval model than previously but without the spatial knowledge of the graphs. The main assumptions are that the structure of the graphs is sufficient to establish an accurate correspondence and that the model graph is

not always fixed with coordinates in the space. We first constructed a heuristic matching procedure that is able to retrieve structures close to the sought model graph. Then, a constraint programming model has been introduced to retrieve the matching between the two graphs. Both algorithms performs well on clean and simple instances even if the constraint programming approach becomes rapidly less efficient on larger ones. However, symmetries in the model and imperfections present in the search graph led to mitigated results with both procedures.

Finally, here is how we could summarize the main lesson provided by the study of our thesis : the quality of the identification of the problem is directly proportional to the quantity of knowledge exploited. That's why the *coordinates dependant* procedure stands out from the others by its modest use of coordinates information while giving impressively reliable results.

Further work

Because *Science never solves a problem without creating ten more - G. Bernard Shaw*, some ideas of further work are now briefly suggested.

Regarding the results of the two coordinates independent approaches, we observed that both of them gave mitigated results. One being maybe a little two greedy and the other being a little too permissive. We believe that a good idea would be to study an hybrid implementation that would be a trade-off between the two approaches. Reusing the good heuristics of the *TALE-like approach* and combining them with a smart backtracking search could indeed be a good idea.

We also would have liked to explore deeper the use of a basic matching composed of multi starting couples. This approach could be used to decompose the problem into smaller submatching-problems. Then a merging procedure could be used to merge the smaller search results and reconstruct a complete matching.

Appendix A

Python modules

In this Appendix, we briefly describe the useful Python modules that we have used to developed the algorithm described in this master thesis. In order to execute our python scripts, it is needed to install them.

A.1 NumPy

NumPy is a well-known module containing a large collections of data structures and functions useful for scientific computing. In our thesis, we used, amongst others, its powerful array structure, its file reader and file writer functions and its different basic types.

A.2 OpenCV

OpenCV is a famous open source computer vision library released under a BSD license. It contains a lot of different data structures to handle images and functions to approximately perform whatever computer vision operation on images.

In this thesis, this library has been widely used to handle the images, perform the preprocessing image operations (blur, threshold, resizing,...), the visualize the results, perform the OCR operations, and so on

A.3 NetworkX

When handling graphs in python, *NetworkX* is the reference package. It allows to manipulate directed or undirected graphs, with or without attributes on nodes and edges. The functions available easily allow to browse the nodes and edges of the graph.

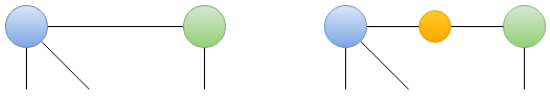
For all the graph matching algorithm, this package has been very helpful to manipulate the model graphs and the search graph.

Appendix B

TALE-like implementation

The pseudo-code B.1 corresponds to the main part of the step 2 of the *TALE-like approach*. It begin by building a priority queue sorted by matching quality from the matches of the step 1. From this queue, it pops the best match to start building the final graph matching. When a couple of nodes is added to the final graph matching, their neighbours are analysed and the possible next matches are added to the queue. The extension of the matching stops when no more match can be found in the nearby nodes.

From a pair of matched nodes, the method `examineNearbyNodes` described at Pseudo-code B.2 analyzes the nearby nodes and try to match them with the procedure `matchNodes`. A nearby node can either be the 1-neighbour or a 2-neighbour that is not already part of the current matching. To restrict or encourage the approximation, this lookahead distance can be reduced or increased. Looking further in the space node's neighbourhood is useful in a graph with irrelevant intermediate nodes. It indeed allows the extension of the match to skip the irrelevant nodes of the 1 or 2-neighbourhood and match a relevant node present in the 3-neighbourhood for example (see figure B.1). Looking further in the model node's neighbourhood allows to keep growing the match even if a corresponding space node does not exist (see figure B.2).



(a) Model graph (b) Space graph

Figure B.1: By trying to match the 1-neighbourhood of the model node with the 2-neighbourhood of the space, a match between the blue and green nodes can be found even if a noisy orange node exists between them in the space graph.

The implementation of the `matchNodes` procedure is available at Pseudo-code B.3.

```
function :  extendMatch(mG, sG, basicMatch)
input :     mG, sG, the model and space graphs. The matching of
   important kinds nodes obtained at step 1, basicMatch.
output :    A final graph match M

  put the matches from basicMatch to a priority queue Q sorted by
   their quality
  while Q is not empty :
    pop best match (mN, sN) from Q
    put (mN, sN) in M
    examineNearbyNodes(mG, sG, mN, sN, M, Q)
  return Q
```

Pseudo-code B.1: Extend match

```
function :  examineNearbyNodes(mG, sG, mN, sN, M, Q)
input :     mG, sG, the model and space graphs. mN a model node
   matched to sN. M contains all the matched nodes find so far and
    Q all the candidates matches to be examined.

  mNB1 = 1-neighbors of mN not matched in M
  mNB2 = 2-neighbors of mN not matched in M
  sNB1 = 1-neighbors of sN not matched in either M or Q
  sNB2 = 2-neighbors of sN not matched in either M or Q
  matchNodes(mG, sG, mNB1, sNB1, M, Q)
  matchNodes(mG, sG, mNB1, sNB2, M, Q)
  matchNodes(mG, sG, mNB2, sNB1, M, Q)
```

Pseudo-code B.2: Examine nearby nodes

(a) Model graph (b) Space graph

Figure B.2: By trying to match the 2-neighbourhood of the model node with the 1-neighbourhood of the space, a match between the blue and the green nodes can be found even if a red node of the model is not found in the space.

```
function :  matchNodes(mG, sG, mNs, sNs, M, Q)
input :     mG, sG, the model and space graphs. mNs a set of model
    nodes, sNs a set of space nodes, M contains all the matched
   nodes find so far and Q all the candidates matches to be
   examined.

  for mN in mNs :
    sN = best mapping for mN in sNs
    if sN == None:
      continue
    if mN not matched in Q :
      put (mN, sN) in Q
      remove sN from sNs
    elif (mN, sN) is a better match than the match of mN in Q:
      remove the existing match from Q
      put (mN, sN) in Q
      remove sN from sNs
```

Pseudo-code B.3: Match nodes

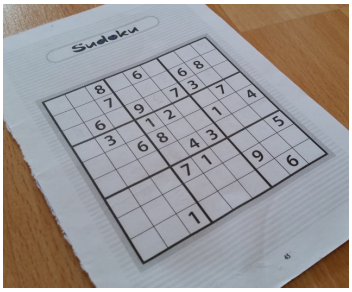
It takes a set of model nodes and a set of space nodes as input, tries to match them and update the priority queue accordingly. For each node in the model set, it evaluates the value of matching this model node with every space node of the set and retains the best one. This best match is then appended to the priority queue if the model node is not already present in the queue. If already present, the match is only appended if is has a greater quality, the less accurate match being removed from the queue.

Appendix C

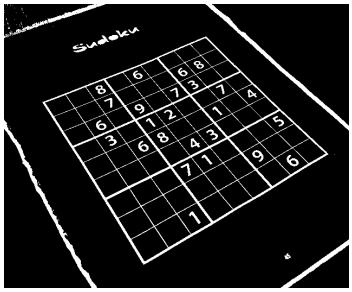
Results

C.1 Coordinates dependent search results

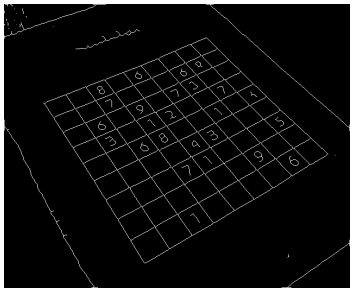
In this section, some results illustrating the whole process of the algorithm using the coordinates dependent search are presented. It briefly summarizes with illustrations the action of each step of the algorithm. The figure C.1 illustrates the key steps of the algorithm.



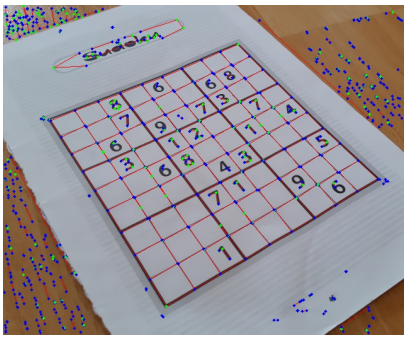
(a) Original image resized



(b) Thresholded image



(c) Thinned image



(d) Search graph extracted

(e) Search graph extracted filtered

Figure C.1: Summary of the key steps of the coordinates dependent search

As we can observe, the image is resized (figure C.1a) and then thresholded (figure C.1b). The thresholded image is a simplification of the coloured image and is required to proceed to the thinning step which will reduce each stroke of the image to become 1-pixel thick only (figure C.1d). Once done, the *search graph extraction process* can begin. Successively, the nodes and then the edges will be discovered (figure C.1e).

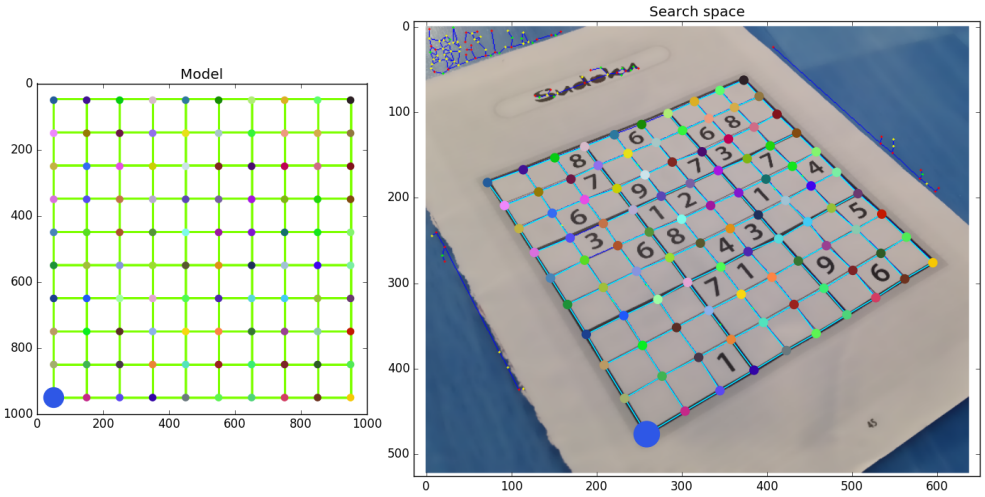


Figure C.2: Matching detected between a sudoku model and the target graph

Finally, given the model graph defined by the user, the algorithm tries to detect and match this model graph with the target graph extracted which leads to the matching visible on the figure C.2.

Let's now illustrate the *modularity* and *generality* of the algorithm. If the desired problem to detect is not a sudoku any more rather a *mathematical pyramid*. In fact it is simply achieved by defining a new model that describes the mathematical pyramid graph. On the figure C.3, we can visualize the result when looking for that new model. Finally, to solve this new problem, only the resolution algorithm must be adapted.

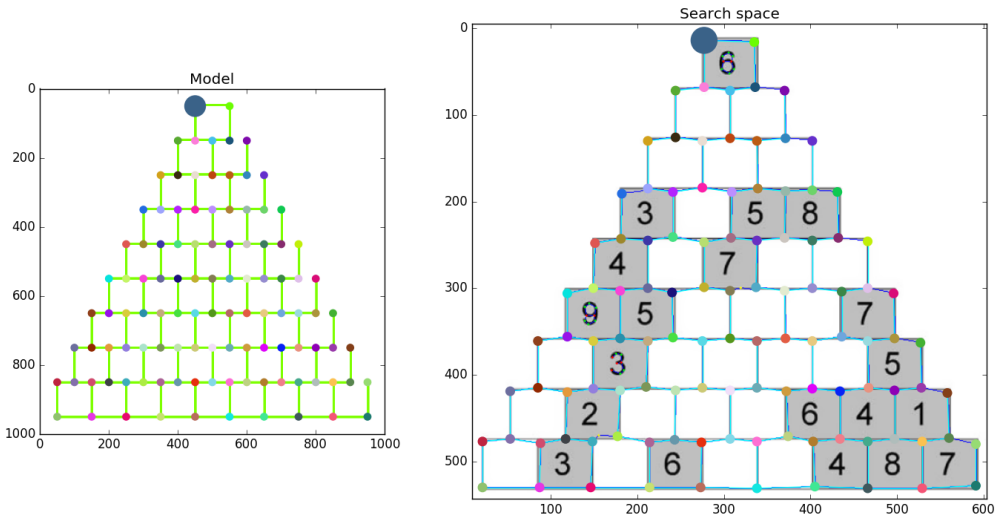


Figure C.3: Matching detected between a mathematical pyramid model and the target graph

C.2 Constraint programming search results

In this section, some results of the constraint programming approach are presented and commented. As said, both constraint programming models behave well on small and perfect instances. As an illustration, the figure C.4 shows the result obtained for both model when trying to match a sudoku grid of size 6 to itself.

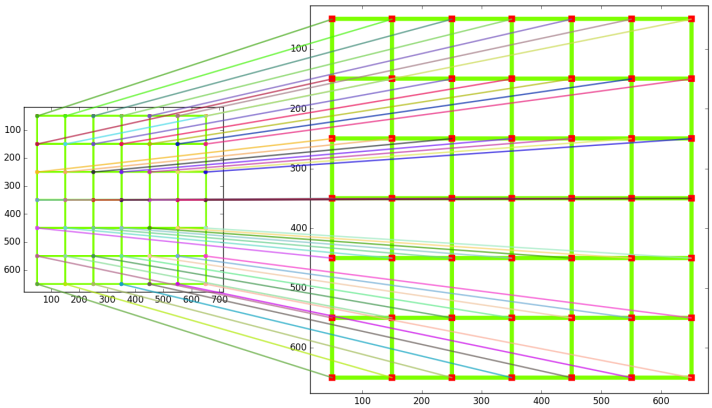


Figure C.4: Matching result of a sudoku of size 6 to itself.

For perfect instances, both models behave correctly until sudoku instances of size 7, sometimes 8. But when it comes to deal with real sudoku grids of size 9 by 9, the symmetries present in the sudoku tend to guide the search into local optimums.

Let's have a look on the figure C.5 that illustrates the result of the naive CP model

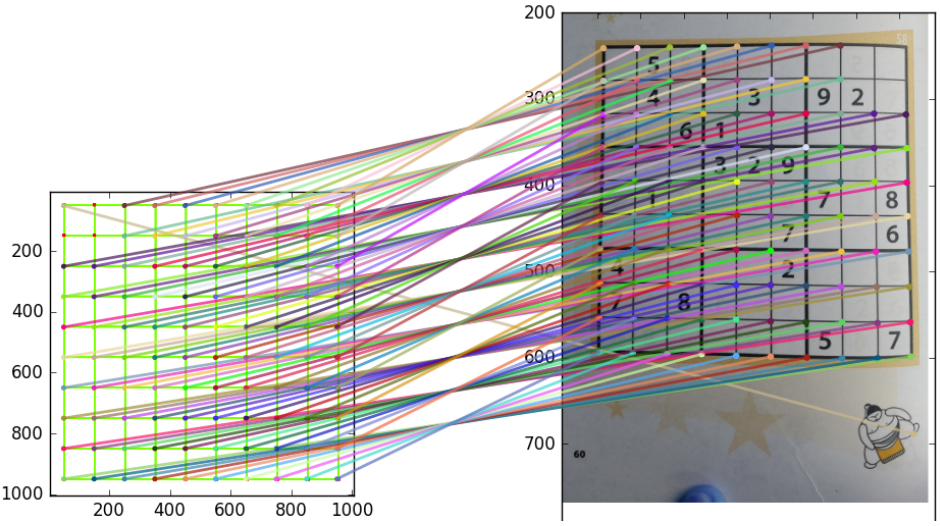


Figure C.5: Matching result of a real sudoku instance with the naive model.

on a real sudoku instance. In this instance, the graph detection step applied before the graph matching search correctly extracted all the 100 nodes and links of the sudoku but also some other noisy nodes.

As can be observed on the figure, the matching seems to be relatively correct. But let's observe the matching of the upper left nodes (see figure C.6).

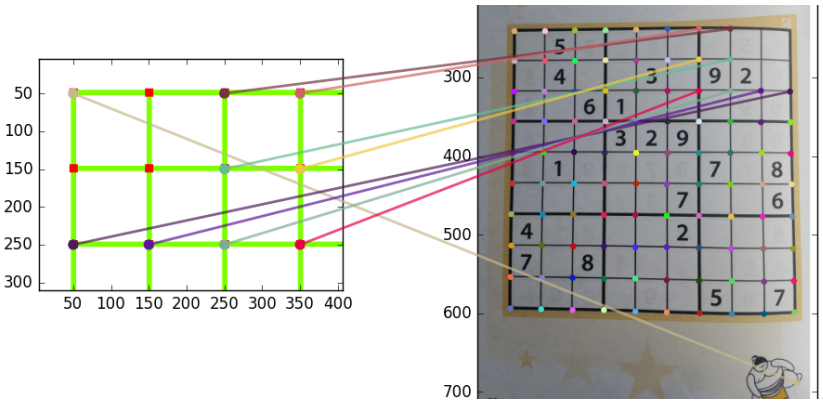


Figure C.6: Matching result of a real sudoku instance with the naive model - Zoom version

What is observed here is that during the LNS search, some of the upper left nodes have been attributed to the dummy value to be missing (this is the case for the model nodes $(1, 0)$, $(0, 1)$ and $(1, 1)$). Due to that, the upper left corner $(0, 0)$ is completely randomly assigned. Some others have been matched in a wrong region of the sudoku. What happens is that that upper left part of the sudoku is mismatched to a wrong region

of the search space while the rest of the model is quite correctly matched. Nevertheless it is imperfect and to improve this matching according the constraints ruling the possible values of missing model nodes, a lot of variables of the current matching should be relaxed. Unfortunately, with LNS, a sufficiently large amount of variables is never relaxed to allow an improvement of the current matching. And relaxing and optimizing a large amount of variables is very costly due to the size of the search. This situation observed here is in fact a local optimum that is very hard to escape when performing an LNS search for the graph matching problem described in this thesis.

C.3 Complexity of the coordinates dependent search

Let's m and s respectively be the *total number of nodes* in the model and space graph. Let $\underline{m}$ and $\underline{s}$ correspond to the total number of edges in the model and search graph respectively. Finally let's denote by m° and s° the maximum degree of model nodes and space nodes respectively.

First we look at the complexity for generating a matching (the `generateMatching` function) starting from a basic match. In the worst case, the algorithm goes through all edges of the model once : $\underline{m}$. For each one, it evaluates if an unexplored space edge matches. Those space edges are selected adjacent to the current space node being analysed. The maximal number of such edges corresponds to the maximal degree of a space node : s°. The complexity of this function is then $\mathcal{O}(\underline{m}s^\circ)$.

This function is called for each basic match tried. The maximal number of such basic matches that the function `getBasicMatchings` may return is then computed. For each couple of space and model node (ms is the total number of couples), all the possible transformations are tried. One transformation exists for each couple of adjacent model and space edges (maximal number of $m^\circ s^\circ$). The total amount of basic matchings the algorithm may return is given by all the possible transformations tried at each possible combination of space and model nodes : $\mathcal{O}(mm^\circ ss^\circ)$. In practice, this amount is never reached since generation of basic matches is sorted so the most reliable ones are returned first.

Bibliography

[1] Laszlo Babai. Graph isomorphism in quasipolynomial time. *University of Chicago*, 2016.

[2] Gary Bradski and Adrien Kaehler. *Learning Computer Vision with the OpenCV Library*. O'Reilly, 2008. ISBN 978-0-596-51613-0.

[3] Stephan A. Cook. The complexity of theorem-proving procedures. *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971. doi: 10.1145/800157.805047.

[4] Nicholas Dahm, Horst Bunke, Terry Caelli, and Yongsheng Gao. Efficient subgraph matching using topological node feature constraints. *Pattern Recognition*, 48:317–330, 2014. doi: 10.1016/j.patcog.2014.05.018.

[5] M. Dirnberger, T. Kehl, and A. Neumann. Nefi: Network extraction from images. *Scientific Reports*, 5, 2015. URL `http://dx.doi.org/10.1038/srep15669`.

[6] Robert Fisher. The evolving, distributed, non-proprietary, on-line compendium of computer vision. URL `http://homepages.inf.ed.ac.uk/rbf/CVonline/CVentry.htm`.

[7] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979. ISBN 0716710447.

[8] OceanColor Biology Processing Group. Resampling methods, Last update in 2016. URL `http://seadas.gsfc.nasa.gov/help/general/ResamplingMethods.html`.

[9] Zicheng Guo and Richard W. Hall. Parallel thinning with two-subiteration algorithms. *Commun. ACM*, 32(3):359–373, March 1989. doi: 10.1145/62065.62074. URL `http://doi.acm.org/10.1145/62065.62074`.

[10] Salim Jouili and Salvatore Tabbone. Advanced concepts for intelligent vision systems: 11th international conference, acivs 2009, bordeaux, france, september 28–october 2, 2009. proceedings. pages 89–99, 2009. doi: 10.1007/978-3-642-04697-1_9. URL http://dx.doi.org/10.1007/978-3-642-04697-1_9.

[11] Salil Kapur and Nisarg Thakkar. *Mastering OpenCV Android Application Programming*. Packt Publishing Ltd., 2015. ISBN 978-1-78398-820-4.

[12] Harish Kumar and Paramjeet Kaur. A comparative study of iterative thinning algorithms for bmp images. *International Journal of Computer Science and Information Technologies*, 2(5):2375–2379, 2011. ISSN 0975-9646.

[13] M. Lades, J. C. Vorbruggen, J. Buhmann, J. Lange, C. von der Malsburg, R. P. Wurtz, and W. Konen. Distortion invariant object recognition in the dynamic link architecture. *IEEE Transactions on Computers*, 42(3):300–311, Mar 1993. ISSN 0018-9340. doi: 10.1109/12.210173.

[14] OscaR Team. OscaR: Scala in OR, 2012. Available from https://bitbucket.org/oscarlib/oscar.

[15] Pramod Kumar Pandey, Yaduvir Singh, and Sweta Tripathi. Article: Image processing using principal component analysis. *International Journal of Computer Applications*, 15(4):37–40, February 2011. Full text available.

[16] U. Ramer. An iterative procedure for the polygonal approximation of plane curves. 1:244–256, 1972.

[17] Intel's research center and a research team in computer vision Itseez. *The OpenCV 2.4 API reference manual*, First edition written in 2011 and continiously updated until now. URL http://docs.opencv.org/2.4/modules/refman.html.

[18] Retargeting.com. Marketing color image. URL https://retargeting.biz/blog/wp-content/uploads/2015/09/Marketing-Colors.jpg.

[19] Michael Rudolf. *Theory and Application of Graph Transformations: 6th International Workshop, TAGT'98, Paderborn, Germany, November 16-20, 1998. Selected Papers*, chapter Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching, pages 238–251. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000. ISBN 978-3-540-46464-8. doi: 10.1007/978-3-540-46464-8_17. URL http://dx.doi.org/10.1007/978-3-540-46464-8_17.

[20] S. Suzuki and K. Abe. Topological structural analysis of digitized binary images by border following. 30:32–46, 1985.

[21] Yuanyuan Tian and Jignesh M. Patel. Tale: A tool for approximate large graph matching. In *Proceedings of the 2008 IEEE 24th International Conference on Data*

Engineering, ICDE '08, pages 963–972, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-1-4244-1836-7. doi: 10.1109/ICDE.2008.4497505. URL `http://dx.doi.org/10.1109/ICDE.2008.4497505`.

[22] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. pages 839–846, Jan 1998. doi: 10.1109/ICCV.1998.710815.

[23] Matthew Turk and Alex Pentland. Eigenfaces for recognition. *Journal of cognitive neuroscience*, 3(1):71–86, 1991.

[24] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, January 1976. ISSN 0004-5411. doi: 10.1145/321921.321925. URL `http://doi.acm.org/10.1145/321921.321925`.

[25] Julian R. Ullmann. Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism. *J. Exp. Algorithmics*, 15:1.6:1.1–1.6:1.64, February 2011. ISSN 1084-6654. doi: 10.1145/1671970.1921702. URL `http://doi.acm.org/10.1145/1671970.1921702`.

[26] Virtual-labs.ac.in. Digital processing lab. URL `http://test.virtual-labs.ac.in/labs/cse19/neigh/convolution.jpg`.

[27] Virginia Vassilevska Williams. Cs267 graph algorithms. University Lecture, 2015.

[28] L. Wiskott, J. M. Fellous, N. Kuiger, and C. von der Malsburg. Face recognition by elastic bunch graph matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(7):775–779, Jul 1997. ISSN 0162-8828. doi: 10.1109/34.598235.

[29] Lei Xu and I. King. A pca approach for fast retrieval of structural patterns in attributed graphs. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 31(5):812–817, Oct 2001. ISSN 1083-4419. doi: 10.1109/3477.956043.

[30] Jian Yang, D. Zhang, A. F. Frangi, and Jing yu Yang. Two-dimensional pca: a new approach to appearance-based face representation and recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(1):131–137, Jan 2004. ISSN 0162-8828. doi: 10.1109/TPAMI.2004.1261097.

[31] Tian Yuanyuan and Patel Jignesh M. Tale: A tool for approximate large graph matching. pages 963–972, 2008. doi: 10.1109/ICDE.2008.4497505. URL `http://dx.doi.org/10.1109/ICDE.2008.4497505`.

[32] Stéphane Zampelli, Yves Deville, and Pierre Dupont. Approximate constrained subgraph matching. page 832–836, 2005.

[33] Stephane Zampelli, Yves Deville, and Pierre Dupont. Declarative approximate graph matching using a constraint approach. 2005.

[34] Stéphane Zampelli, Yves Deville, and Christine Solnon. Solving subgraph isomorphism problems with constraint programming. *Constraints*, 15(3):327–353, 2010. ISSN 1572-9354. doi: 10.1007/s10601-009-9074-3. URL `http://dx.doi.org/10.1007/s10601-009-9074-3`.

[35] B. Zhang and J. P. Allebach. Adaptive bilateral filter for sharpness enhancement and noise removal. *IEEE Transactions on Image Processing*, 17(5):664–678, May 2008. ISSN 1057-7149. doi: 10.1109/TIP.2008.919949.

[36] T. Y. Zhang and C. Y. Suen. A fast parallel algorithm for thinning digital patterns. *Commun. ACM*, 27(3):236–239, March 1984. doi: 10.1145/357994.358023. URL `http://doi.acm.org/10.1145/357994.358023`.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**