On the Compression of SVG Images

Sander Ginn
sander@ginn.it

November 9, 2017, 52 pages

Academic supervisor:	dr. Clemens Grelck
Host supervisor:	Rolf Timmermans
Host organisation:	Voormedia, http://www.voormedia.com

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Contents

Abstract

Compression of raster-based images is a widely researched field. Compression of their vector-based counterparts is not, as their scale invariance cannot be compromised. SVG is the most common type of vector-based images. The amount of data points used to describe an SVG image can be redundant, making the image larger than necessary.

We developed a six-step approach that reduces the file size by simplifying curves in the image. The approach was implemented in a prototype. A set of test images were used to evaluate the prototype with varying parameters for the conversion steps. The output images were then compared to the input file by calculating the SSIM index metric.

The results indicated that the method produces good results for images with specific traits. Images with a lot of curves yielded the best results. In contrast, images with a lot of straight lines failed often. The results show that compression of SVG images is possible to at least some extent. We conclude that a one-size-fits-all solution is not yet reasonable. Future work should focus on improving the results of images with a lot of straight lines and sharp corners.

Acknowledgements

I would like to thank Rolf for his continuous support and input, Clemens for the feedback on my academic approach, and Voormedia for facilitating my research. Furthermore, my appreciation goes to the entire Voormedia team for the enjoyable time I spent in the office. Finally, I would like to extend my sincere gratitude to Janelle for making my life better in so many ways.

Chapter 1

Introduction

This chapter introduces the problem domain. Section 1.1 explains the basics of image compression. In Section 1.2, the scalable vector graphics format is discussed. A brief introduction of the host company is given in Section 1.3. Section 1.4 states the research problem, and Section 1.5 lays out the research questions. Finally, a thesis outline is provided in Section 1.6.

1.1 Image compression

Image compression is the act of compressing the data that describes an image. It mainly serves to minimise storage and transfer costs. Lossless compression applies techniques to compress the data in such a way that the image remains identical after compression. It usually results in marginal compression results. In contrast, lossy compression approximates and discards data in order to achieve reduced data size. The quality of an image can often be compromised to an extent where the fidelity of the image remains sufficient for the intended purpose while reducing the file size dramatically. This is desirable for use cases such as websites: smaller images load faster and thus reduce loading time of a page, which is a major influence on the perceived quality of a website [24].

Existing image compression methods such as JPEG and PNG apply to raster-based graphics. Raster-based images have predefined dimensions which compose the image raster. Each pixel in the raster is assigned a color value. Lossy compression methods use one or more techniques to remove data per pixel or group of pixels. JPEG, for example, applies a discrete cosine transform to blocks of 8×8 pixels to separate frequencies in each block. It then discards the least important frequencies through quantisation. The remaining frequencies are collected in the output file and are Huffman-encoded [6].

1.2 Scalable Vector Graphics

An emerging image format is Scalable Vector Graphics (SVG). An SVG image is, as the name suggests, comprised of vectors rather than pixels. The main advantage of vector-based graphics is that it is scale invariant, as the image is rasterised by the client that displays it based on the raster features of the display. If the image is zoomed in or out it is rasterised again, and so the image is crisp at all times [9]. Furthermore, SVG is an extension of the Extensible Markup Language (XML) and compatible with HTML. These features make SVG highly suitable for web use, since it facilitates responsive web design and is supported by all modern browsers by default. Design elements such as logos are often SVG images, since they are rendered on a large variety of dimensions.

1.3 Host company

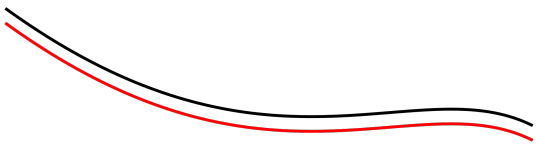
Voormedia B.V.[1] is an Amsterdam based company specialised in digital media production. Two of their core products include TinyPNG[2] and TinyJPG[3]. These products lead their respective markets in image compression due to the user friendly nature: their algorithms automatically determine the optimal compression parameters to ensure the best ratio of size reduction to image quality. As a result, their in-house knowledge on image compression is very rich, and they are interested in exploring future ventures to expand this knowledge in terms of SVG compression.

1.4 Problem statement

Due to the scalable nature of SVG, applying some form of compression to these types of images sounds counter-intuitive at first. After all, the traditional concept of image compression limits how much an image can be scaled before the quality of the image becomes unacceptable, and a key feature of SVG is limitless scaling.

However, since SVG is an extension of XML, and therefore defined by tags and text, we can treat the file as such: a text file. By default, SVG supports gzip compression, which has demonstrated file size reductions of up to 85% [9]. Although the size reduction can be significant, it does not account for other factors that contribute to a larger file size. The main area of improvement not addressed by gzip is redundant and poorly structured data. For example, redundancy occurs in SVG files exported by Adobe Illustrator[4], one of the leading vector drawing applications. They can contain metadata specific to the application which allows all editor traits such as layers and brushes to be preserved when the file is reopened. This metadata significantly increases the file size without changing the visual appearance of the image [14]. An example of poorly structured data is the unnecessary use of `<g>` tags, which define groups of elements to which filters and display styles can be applied. Group tags can be used without actually defining any properties for it, much like the `<div>` tag in HTML, which does not alter the visual appearance, but increases the file size.

Since the aforementioned issues do not impact the appearance of the image itself, addressing them should be seen as a form of lossless compression. Previous work has been done in this field, which will be used in this research. Section 3.7 will describe the specifics of this work. One of the main areas of focus of this research is the feasibility of lossy compression of SVG files. A very simple motivational example is shown in Figure 1.1a. The black line is the result of the first `<path>` element and the red line corresponds to the second `<path>` element in Figure 1.1b. The lines are visually identical, but the red line is composed of two Bézier curve elements, while the black line uses only one. Thus, in this case the red line could theoretically be compressed without any loss of quality. The assumption that drives this research is that more complex SVG images contain lines that are 'overdefined'; lines that can be represented by less data points with an acceptible loss of fidelity.



(a) Different paths but identical lines

```
<svg xmlns="http://www.w3.org/2000/svg" width="2000"
    height="750" version="1.1">

    <path d="M100,100 C 1000,750 1500,300 1900,500"
        style="stroke:black; fill:none; stroke-width:10px;"/>

    <path d="M100,100 C 550,425 900,475 1187.5,468.75 C 1475,462.5
        1700,400 1900,500" style="stroke:red; fill:none;
        stroke-width:10px;" transform="translate(0,50)"/>
</svg>
```

(b) SVG data

Figure 1.1: An example of how two distinct paths produce the same result

[1] http://voormedia.com
[2] http://tinypng.com
[3] http://tinyjpg.com
[4] http://www.adobe.com/products/illustrator.html

1.5 Research questions

We aim to establish the feasibility of SVG compression at the expense of quality, thus the first research question is:

RQ 1. *Can lossy compression be applied to SVG images, making the reduction in fidelity worthwhile?*

To answer RQ 1, it is necessary to define a 'worthwile reduction of fidelity'. Since Voormedia has extensive knowledge in image compression, we will use their knowledge to answer research question 2:

RQ 2. *When is a reduction in fidelity worthwhile?*

Like the other compression tools offered by Voormedia, it is desirable that the optimal algorithm parameters are determined automatically. Therefore, the third research question is:

RQ 3. *Can the algorithm parameters be approximated for optimal compression results?*

While the fundamental difference between raster-based and vector-based images prevents implementation-specific knowledge from being reused, it is likely that Voormedia has other, more general knowledge available that can be applied to this research. Similar to RQ 2, the last research question is:

RQ 4. *How can existing knowledge on image compression be applied to SVG?*

1.6 Thesis outline

Chapter 2 provides background regarding Bézier curves, the SVG file format and line simplification. Chapter 3 discusses the implementation of the prototype. In Chapter 4, the experiment design is laid out. The results of the experiment are reported in Chapter 5. We then discuss the results and explore future work in Chapter 6. Finally, Chapter 7 concludes this thesis.

Chapter 2

Background

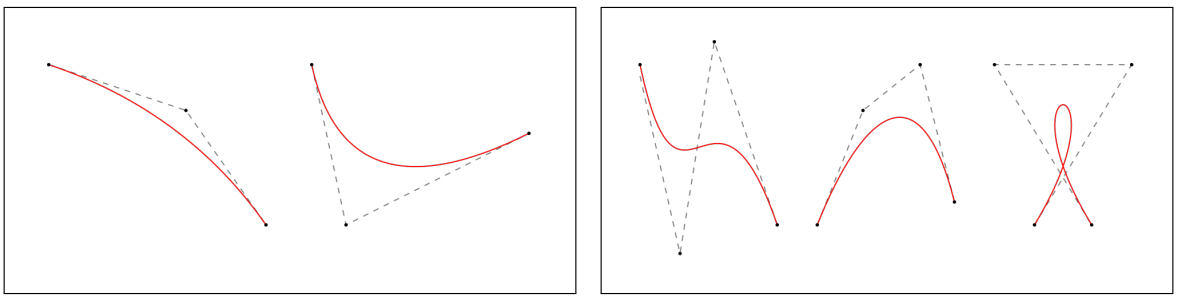
This chapter provides background information about fundamental concepts that play a central role in this research. First, Section 2.1 explains how Bézier curves work. Next, Section 2.2 describes the SVG file format. Finally, related work is outlined in Section 2.3.

2.1 Bézier curves

In the early 1960's, Pierre Bézier developed a method of curve formulation while working at Renault. The goal was to simplify the process of shape design, which was a time consuming process at the time. The result was the Bézier curve, a curve formula that was both intuitive to use and did not require extensive knowledge of mathematics [33].

A Bézier curve is defined by a polygon that is referred to as the Bézier polygon. Perhaps a more suitable name would have been the Bézier polyline, as it is not required that the polygon is closed. The polygon consists of a start and an end point, combined with a number of control points. The number of control points does not have a limit. However, there are two Bézier curves that are used most frequently: the quadratic Bézier curve is drawn using one control point, and the cubic Bézier curve uses two control points. Examples of each type are shown in Figure 2.1. When more control points are used, the calculation process does not scale well timewise, and therefore, more complex curves are usually defined by chaining quadratic and cubic Bézier curves [3].

Coincidentally, Paul de Casteljau developed a very similar solution a few years before while working for Citroën. Because Citroën kept his findings confidential, Bézier's work received more credit. However, the computation of points on the Bézier curve for the corresponding polygon is most effective through the algorithm developed by De Casteljau. This algorithm is also part of our proposed solution. This method of point calculation is described in Section 3.4.



(a) Quadratic Bézier curves

(b) Cubic Bézier curves

Figure 2.1: Examples of the two most prevalent Bézier curves. The dashed line represents the Bézier polygon

2.2 Scalable Vector Graphics

The SVG file format is an open standard developed by the World Wide Web Consortium (W3C). The SVG Working Group was founded in 1998 out of a growing concern about the shortcomings of HTML with respect to proprietary graphics technologies [28]. These image types required workarounds to be used in combination with HTML, leading to undesirable design patterns. In order to address this issue, the W3C invited industry leaders to collaborate in the development of an open, patent-free standard for vector graphics. At the same time as the SVG Working Group formed, the Extensible Markup Language (XML) was quickly becoming the de facto standard for encoding documents in a manner that is easy to interpret by both humans and machines. It is highly favored for specific applications. Due to its concise but strict set of syntax rules, parsing XML extensions is independent of its grammar [22]. Furthermore, XML integrates seamlessly with Cascading Style Sheets (CSS) for optimal reuse of style elements in of SVG images.

Due to these properties, the SVG Working Group decided to opt for an extension of XML as the new file type. Out of the submissions that they received from the group members, four incorporated XML in their proposals [13]:

- **Web Schematics**, proposed by CCLRC
- **Precision Graphics Markup Language (PGML)**, proposed by Adobe, IBM, Netscape, and Sun Microsystems
- **Vector Markup Language (VML)**, proposed by Autodesk, Hewlett-Packard, Macromedia, and Microsoft
- **DrawML**, proposed by Excosoft

Out of these four submissions PGML and VML were the top contenders, but both had their own issues, and so the decision was made to take the best of both submissions to form the recommendation for SVG 1.0 [23]. Newer versions of SVG focus on incorporating new versions of web features such as HTML [10].

The latest specification, SVG 1.1, describes a number of areas that together compose the suite of SVG functionality. We outline the areas that are relevant to this research in this section. Filter effects [41] are not described as it is a rather complex domain and remain unaltered in this research; other areas pertain to interactive web use such as animated and scripted SVG images.

Structure elements A number of core elements are part of SVG that are used to introduce the necessary structure to an SVG image [40]. Without these elements, styling the SVG would be a very cumbersome task. Aside from this, several elements for accessibility are implemented so that the data can be at least partially meaningful in situations where the image can not be rendered or is interpreted by machines. The relevant elements are detailed in Figure 2.2.

Basic shapes SVG contains a set of 6 basic shapes which can be used with their respective tags [38]. Figure 2.3 defines these shapes.

Paths Aside from the basic shapes SVG also implements the path element, denoted by the `<path>` tag [45]. In essence, the path element is used to create any shape that is not covered by the basic shapes. A path element has only one mandatory attribute, `d`, which contains the path data. The path string that belongs to this attribute is extremely flexible in how it can be defined. An example of this flexibility is shown in Figure 2.4: these paths all produce the same two lines through different syntax. For example, the spaces between the coordinates pertaining to the last issued command on line 2 can be replaced by commas; the space between the last coordinate and new command can be ommitted; if the next path component is of the same type as the previous then the command for the new component can also be ommitted, as on line 4; if the command is capitalised the coordinates are absolute, whereas lowercase means relative to the last position. These are just a few examples of the versatility of the path data, and they can also be used in

Tag	Unique attributes	Description
<svg>	version, width, height	the root tag of an SVG file, containing information about the entire document
<g>	-	a container element that groups the inner elements together, allowing all elements to be styled at once and enables reuse of objects
<defs>	-	elements defined inside this tag are not rendered until they are referenced later, which is useful for structuring properties such as filters and masks
<desc>/<title>	-	descriptors for accessibility purposes, such as screen readers
<use>	x, y, width, height	references the id of an <svg>, <g> or other <use> element in order to render the same element and its children at the provided coordinates

Figure 2.2: Structure elements of SVG

Tag	Unique attributes	Description
<rect>	x, y, width, height	draws a rectangle starting at (x, y) with the specified width and height
<circle>	cx, cy, r	draws a circle with (cx,cy) as the center point and radius r
<ellipse>	cx, cy, rx, ry	draws an ellipse with (cx,cy) as the center point, rx as the horizontal radius, and ry as the vertical radius
<line>	x1, y1, x2, y2	draws a straight line from (x1,y1) to (x2,y2)
<polyline>	points	draws a sequence of straight lines with the coordinate pairs in points
<polygon>	points	draws a closed shape composed of a set of connected lines with the coordinate pairs in points

Figure 2.3: The basic shapes of SVG

combination. While this can be beneficial, it also increases the complexity of parsing the data. The complete set of commands that can be used in a path element is described in Figure 2.5, and simple examples of each command are illustrated in Figure 2.6.

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1">
    <path d="M10 10 L 100 100 L 150 150" />
    <path d="M10,10L100,100L150,150" />
    <path d="M10 10L100 100 150 150" />
    <path d="M10 10l90 90 50 50" />
</svg>
```

Figure 2.4: Syntactically different, semantically identical paths

Text SVG has a dedicated text element that uses the <text> tag to render the text content within the tags [46]. It is rendered as XML character data, which means it is treated as any other text. Hence, it can be searched for, highlighted, and indexed by search engines. The text is rendered at the provided (x,y) parameters. A single text element can have multiple stylings attached to it by defining span elements with <tspan>.

Token	Parameters	Description
M/m	`x y`	moves the point to `(x,y)` to start a new sub-path
Z/z	-	closes the sub-path by drawing a straight line to the path's initial point
L/l	`x y`	draws a line to `(x,y)`
H/h	`x`	draws a horizontal line to the point with `x` and unchanged `y`
V/v	`y`	draws a vertical line from to the point with unchanged `x` and `y`
C/c	`x1 y1 x2 y2 x y`	draws a cubic Bézier curve to `(x,y)` with control points `(x1,y1)` and `(x2,y2)`
S/s	`x2 y2 x y`	draws a smooth cubic Bézier curve to `(x,y)` where the first control point is equal to the second control point of the preceding C or S command
Q/q	`x1 y1 x y`	draws a quadratic Bézier curve to `(x,y)` with control point `(x1,y1)`
T/t	`x y`	draws a smooth quadratic Bézier curve to `(x,y)` where the control point is equal to the control point of the preceding Q or T command
A/a	`rx ry x-axis-rotation large-arc-flag sweep-flag x y`	draws an elliptical arc to `(x,y)` with radii `(rx,ry)`. `x-axis-rotation` indicates how the ellipse is rotated. `large-arc-flag` determines whether the long or short path between `(rx,ry)` and `sweep-flag` states whether a negative or positive drawing angle is used

Figure 2.5: Path commands

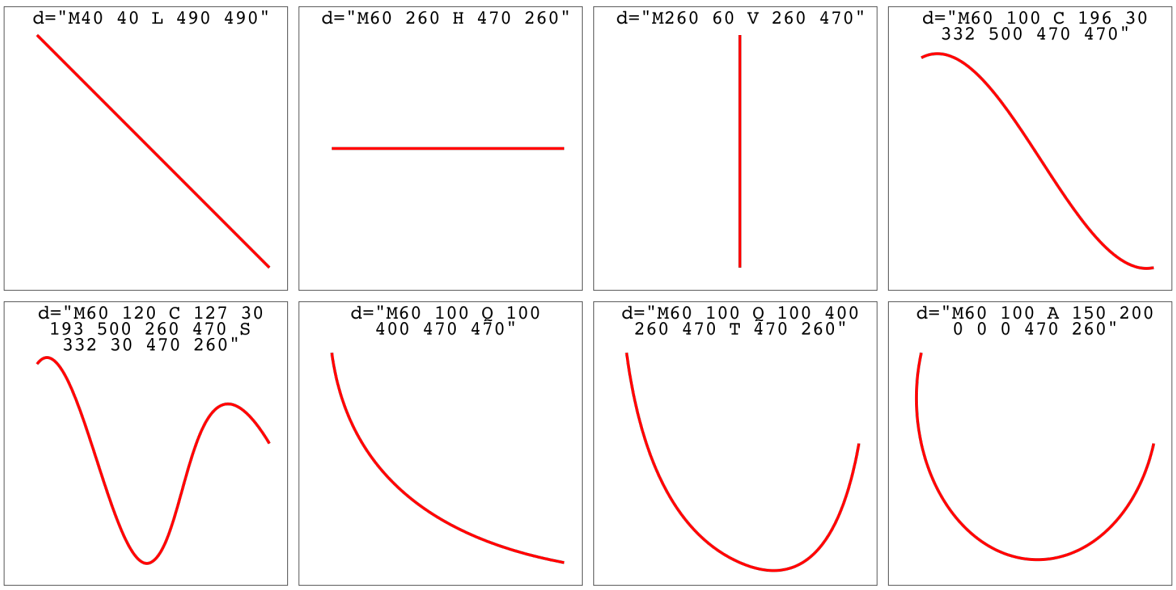


Figure 2.6: From left to right, top to bottom: output for tokens `L`, `H`, `V`, `C`, `S`, `Q`, `T`, `A`. The header of each example contains the path data string

Gradients and patterns Shape, path, and text elements can be filled or stroked with a gradient, color or pattern [42]. SVG implements two types of gradients: the `<linearGradient>` and `<radialGradient>`. The linear gradient moves in a linear direction, gradually shifting between the defined colors. The radial gradient is a circular gradient, moving outwards of a defined center

point. The actual colors and their offsets are defined in `<stop>` elements within the gradient tag. The `<pattern>` element is in essence its own canvas where the pattern can be designed with any functionality that SVG supports.

Clipping and masking SVG supports clipping paths and masks through the `<clipPath>` and `<mask>` tags [39]. A clipping path defines an outline that hides any elements not within its borders. A mask is more detailed than a clipping path by allowing different properties to be defined within the masked area, such as partial opacity.

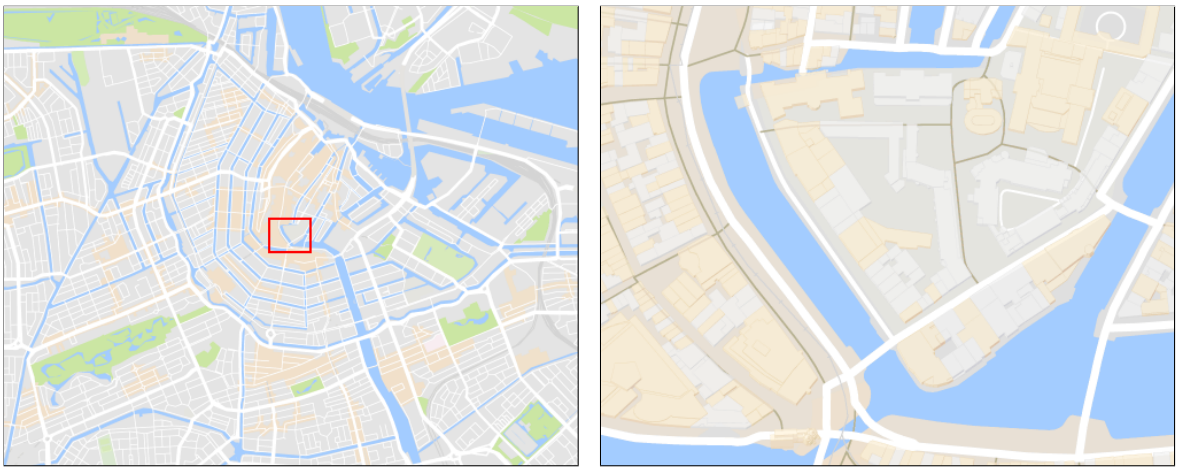
2.3 GIS and line simplification

Although no comparable research has been conducted which relates directly to SVG compression, a part of this research builds upon a technique that is often used in geographic information systems (GIS). These systems are designed to represent geographic data and allow the user to zoom in and out, so the desired amount of data is viewed. As the user zooms in, the data can be displayed in greater detail and vice versa on zooming out. This requires the system to dynamically adjust the fidelity of geographic data. Many GISs take a vector-based approach to tackle this problem [25].

One particular technique that is commonplace in these systems is line simplifcation [7, 30]. In the context of a GIS, this is used to determine with which precision lines should be displayed based on the zoom level. As Figure 2.7a shows, the lines of the canal within the red square do not display the same level of detail as the same area in Figure 2.7b. This is a result of applying line simplification to the data points that describe the contours of the canal. This technique can be used to address the problem of 'overdefined' lines, as described in Section 1.4.

The process of simplifying lines is applied in a variety of fields, aside from GISs. In the field of computer vision, line simplification is used to filter superfluous data from sensors in order to speed up recognition algorithms and to cope with noisy outliers [16]. In computer graphics, it finds its use in cases like video games and flight simulators [15].

In an evaluation of line simplification algorithms, Shi and Cheung analysed 9 different line simplification algorithms [34]. The algorithms were evaluated based on the visual difference, displacement, shape distortion, and computation time using the results of simplifying a set of test cases. In terms of visual difference, the Ramer-Douglas-Peucker algorithm proved to be the most accurate. However, at the same time, it is one of the most time consuming algorithms. Because speed is not a primary concern in this research, we have opted to use the Ramer-Douglas-Peucker algorithm for our prototype. The algorithm is described in detail in Section 3.5.



(a) Zoomed out view of Amsterdam's center

(b) Zoomed in view of Amsterdam's center

Figure 2.7: Different zoom levels demonstrating line simplification

Chapter 3

Prototype design

This chapter outlines the implementation of the prototype. First, Section 3.1 describes the hypothesis behind our solution and the intended goals of the prototype. Section 3.2 discusses what steps are taken to achieve the intended goals. Sections 3.3 to 3.8 describe the steps in greater detail. Finally, Section 3.9 briefly discusses what technology is used in the prototype.

3.1 Solution hypothesis

As discussed in Section 1.4, the hypothesis being tested in this research is that SVG files can contain lines that are 'overdefined'. We assume that the number of data points that describe these lines can be reduced while maintaining an acceptable level of image fidelity. Section 4.2 defines what is deemed acceptable and how it is validated.

3.2 Prototype description

To achieve the desired effect as described in Section 3.1, we implement a prototype that applies a number of algorithms to an input SVG file. The algorithm takes a 6-step approach of preparation, compression, and finalisation. We show a pipeline view of the prototype in Figure 3.1. First, we prepare the input file by converting arcs to polylines. Additionally, paths are converted to polylines with De Casteljau's algorithm. Next, all line elements are simplified with the Ramer-Douglas-Peucker algorithm to remove redundant data. The line elements are then evaluated for conversion to cubic Bézier curves. In some cases, lines are left as is, since it makes no sense to convert straight lines to Bézier curves. Furthermore, acute angles are preserved through two line elements.

With respect to lossless compression, we evaluated several tools that serve the same purpose in how they approach the problem [35, 5]. It quickly became apparent that these tasks are quite trivial in terms of complexity but significant in terms of implementation time. Taking time constraints and the scope of this research in consideration, we decided to reuse these existing tools rather than reinventing the wheel. Should the result of this research be developed further, this part of the prototype pipeline will be implemented in a bespoke module. To achieve maximum compression results, the final result is gzipped.

3.3 Arc to cubic Bézier conversion

The first processing step in the pipeline converts arc elements in each path to one or more cubic Bézier curves. While other shapes such as circles or rectangles are left as is, arc elements are converted as they can be reconstructed with Bézier curves efficiently. Especially when the arc is preceded and succeeded by other elements (in other words, not the first or last path element), the arc can 'overflow' into the neighbouring Bézier elements. The conversion is performed by the set of equations reported in this section [43].

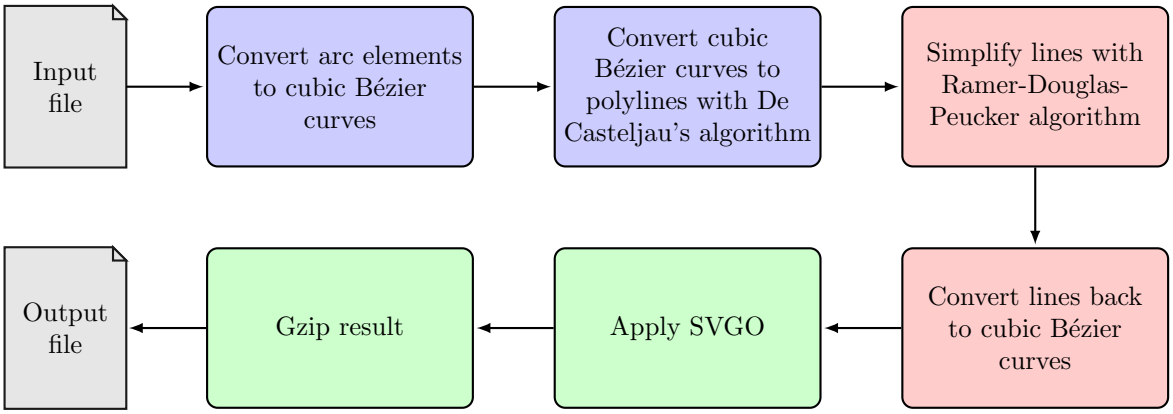


Figure 3.1: SVG processing pipeline. The blue nodes indicate preparatory steps, the red nodes indicate lossy compression steps, and the green nodes indicate lossless compression steps

Figure 3.2 describes the set of parameters that are used in the arc command and how they will be referred to in the equations.

Symbol	**Description**
(x_1, y_1)	the coordinates of the current point on the path
r_x and r_y	the radii of the ellipse
ϕ	angle from the x-axis on the current coordinate system to the x-axis of the ellipse
f_A	large arc flag, which for arc span α is 0 if $\alpha \leq 180°$ or 1 if $\alpha > 180°$
f_S	sweep flag, which is 0 if the arc is drawn along the inner circle, and 1 if it is drawn along the outer circle
(x_2, y_2)	the coordinates of the final point of the arc

Figure 3.2: Arc symbols and parameters

First, the origin is normalised and any rotation is removed with (3.1):

$$\begin{pmatrix} {x_1}' \\ {y_1}' \end{pmatrix} = \begin{pmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{pmatrix} \cdot \begin{pmatrix} \frac{x_1 - x_2}{2} \\ \frac{y_1 - y_2}{2} \end{pmatrix} \tag{3.1}$$

We then correct radii that are out of range. If the radii do not meet the constraints in (3.2), then this is a straight line from (x_1, y_1) to (x_2, y_2) and we stop. Otherwise, we take the absolute values of the radii and calculate the lambda value with (3.3) using the results from (3.1).

$$r_x \neq 0, \quad r_y \neq 0 \tag{3.2}$$

$$\Lambda = \frac{({x_1}')^2}{|r_x|^2} + \frac{({y_1}')^2}{|r_y|^2} \tag{3.3}$$

If $\Lambda \leq 1$, the radii need no further corrections. If $\Lambda > 1$, then the radii are corrected with (3.4).

$$r_x \ = \sqrt{\Lambda} r_x, \quad r_y \ = \sqrt{\Lambda} r_y \tag{3.4}$$

The next step calculates the coordinates which represent the center of the arc in the adjusted coordinate system with (3.5) and transforms it back to the original coordinate system with (3.6).

$$\begin{pmatrix} {c_x}' \\ {c_y}' \end{pmatrix} = \pm\sqrt{\frac{r_x^2 r_y^2 - r_x^2({y_1}')^2 - r_y^2({x_1}')^2}{r_x^2({y_1}')^2 + r_y^2({x_1}')^2}} \begin{pmatrix} \frac{r_x {y_1}'}{r_y} \\ -\frac{r_y {x_1}'}{r_x} \end{pmatrix} \tag{3.5}$$

$$\begin{pmatrix} c_x \\ c_y \end{pmatrix} = \begin{pmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{pmatrix} \cdot \begin{pmatrix} {c_x}' \\ {c_y}' \end{pmatrix} + \begin{pmatrix} \frac{x_1 + x_2}{2} \\ \frac{y_1 + y_2}{2} \end{pmatrix} \tag{3.6}$$

The following step is to compute θ_1 and $\Delta\theta$. To do so, we need to calculate the angle between two vectors with (3.7), applying it to (3.8) and (3.9).

$$\angle\Big(\vec{u}, \vec{v}\Big) = \pm \arccos \frac{\vec{u} \cdot \vec{v}}{||\vec{u}||\ ||\vec{v}||} \text{ with } \pm \text{ equal to the sign of } u_x v_y - u_y v_x \tag{3.7}$$

$$\theta_1 = \angle \left(\begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} \frac{{x_1}' - {c_x}'}{r_x} \\ \frac{{y_1}' - {c_y}'}{r_y} \end{pmatrix} \right) \tag{3.8}$$

$$\Delta\theta \equiv \angle \left(\begin{pmatrix} \frac{{x_1}' - {c_x}'}{r_x} \\ \frac{{y_1}' - {c_y}'}{r_y} \end{pmatrix}, \begin{pmatrix} \frac{-{x_1}' - {c_x}'}{r_x} \\ \frac{-{y_1}' - {c_y}'}{r_y} \end{pmatrix} \right) \mod 360° \tag{3.9}$$

We then use the results of (3.8) and (3.9) to approximate the control points for each segment with (3.10) [29], where $\{p_1, p_2, p_3, p_4\}$ correspond to the start, first control, second control and end points. As a final step, we need to transform the unit arc points back to the original coordinate system with (3.11) and the results of (3.4) and (3.6).

$$\begin{aligned} \alpha &= \frac{4}{3}\tan\frac{\Delta\theta}{4} \\ (x_1, y_1) &= (\cos\theta_1, \sin\theta_1) \\ (x_4, y_4) &= (\cos(\theta_1 + \Delta\theta), \sin(\theta_1 + \Delta\theta)) \\ (x_2, y_2) &= (x_1 - (y_1\alpha), y_1 + (x_1\alpha)) \\ (x_3, y_3) &= (x_4 + (y_4\alpha), y_4 - (x_4\alpha)) \end{aligned} \tag{3.10}$$

$$\begin{aligned} x = xr_x, &\quad y = yr_y \\ x_p &= (\cos\phi)x - (\sin\phi)y \\ y_p &= (\sin\phi)x + (\cos\phi)y \\ x = x_p + c_x, &\quad y = y_p + c_y \end{aligned} \tag{3.11}$$

After these operations an arc is transformed into one or more cubic Bézier curves in the form of a start and end point and two control points, which is the same notation SVG uses.

3.4 De Casteljau's algorithm

The next processing step converts all Bézier curves to a sequence of lines. To achieve this, we use De Casteljau's algorithm [4]. The algorithm works as follows, with control points $\{p_0, p_1, p_2, p_3\}$:

- Pick a range of values V with the constraint $0 \leq t \leq 1$. A larger quantity of values will result in a smoother curve.

- For each value $t \in V$, do
  - split the line segments $\overline{p_0p_1}$, $\overline{p_1p_2}$ and $\overline{p_2p_3}$ with $p_{1n} = (1-t)p_n + tp_m$;
  - for the new line segments $\overline{p_{10}p_{11}}$ and $\overline{p_{11}p_{12}}$, split the line segment again with $p_{2n} = (1-t)p_{1n} + tp_{1m}$;
  - split the last line segment $\overline{p_{20}p_{21}}$ with $p_t = (1-t)p_{20} + tp_{21}$.
- The set $T = \{p_{t1}, p_{t2}, \dots\}$ contains the points that can be connected to approximate the Bézier curve.

Figure 3.3 illustrates the result for $t = \{0.25, 0.5, 0.75\}$ for a set of control points $\{p_0, p_1, p_2, p_3\}$. Pseudocode of the algorithm is shown in Figure 3.4.

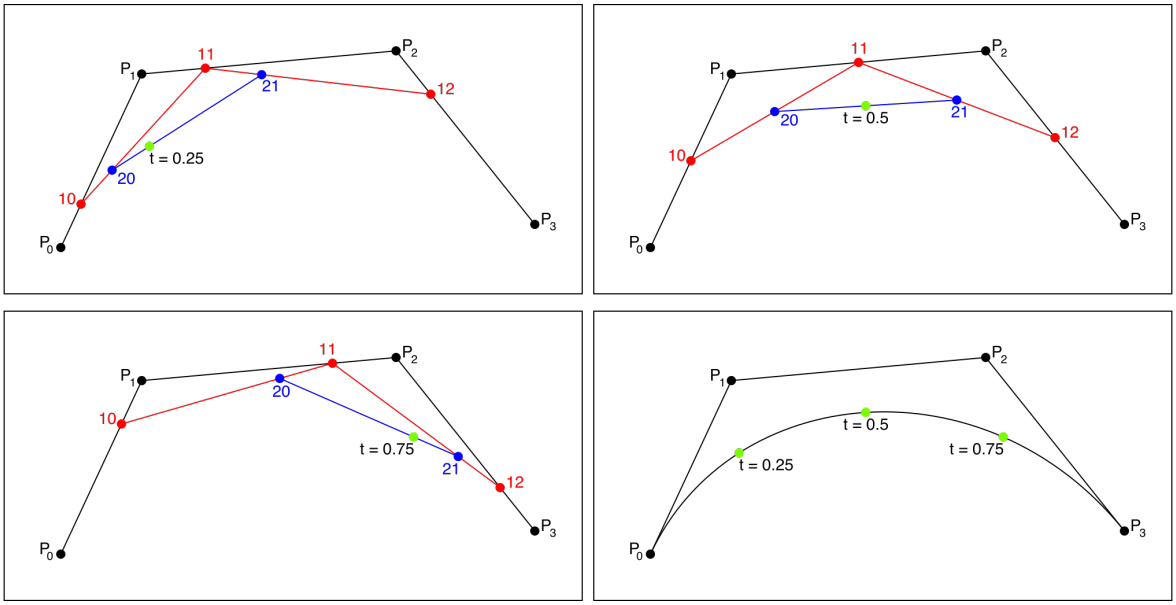


Figure 3.3: An illustration of De Casteljau's algorithm. P_0-P_3 are the Bézier control points. The red and blue points are the result of the first and second iteration respectively. The green point denotes the final value on the curve for a given t

```
1:  function DC(points, t)
2:      temp_points ← points
3:      last ← length(points)−1
4:      for k ← 1 to last do
5:          for i ← 0 to n − k do
6:              temp_points[i] ← (1 − t) ∗ temp_points[i] + t ∗ temp_points[i+1]
7:          end for
8:      end for
9:      return temp_points[0]
10: end function
```

Figure 3.4: Pseudocode for De Casteljau's algorithm

3.5 Ramer-Douglas-Peucker algorithm

The previous step generates a vast amount of data, of which the majority can be discarded. To achieve this, we apply the Ramer-Douglas-Peucker algorithm, a divide-and-conquer algorithm for line

simplification that uses a threshold to determine whether or not a data point should be discarded [12]. The algorithm works as follows, for a threshold ϵ:

- Draw a temporary line between the first and last point in the dataset.
- Find the point of which shortest distance δ to the line of step 1 is the furthest away.
- Two cases can now be distinguished:
  - $\delta < \epsilon$: all points that are not marked to be kept in between the two points are discarded from the data set. The algorithm recurses with the current end point as the starting point and the line's last point as end point;

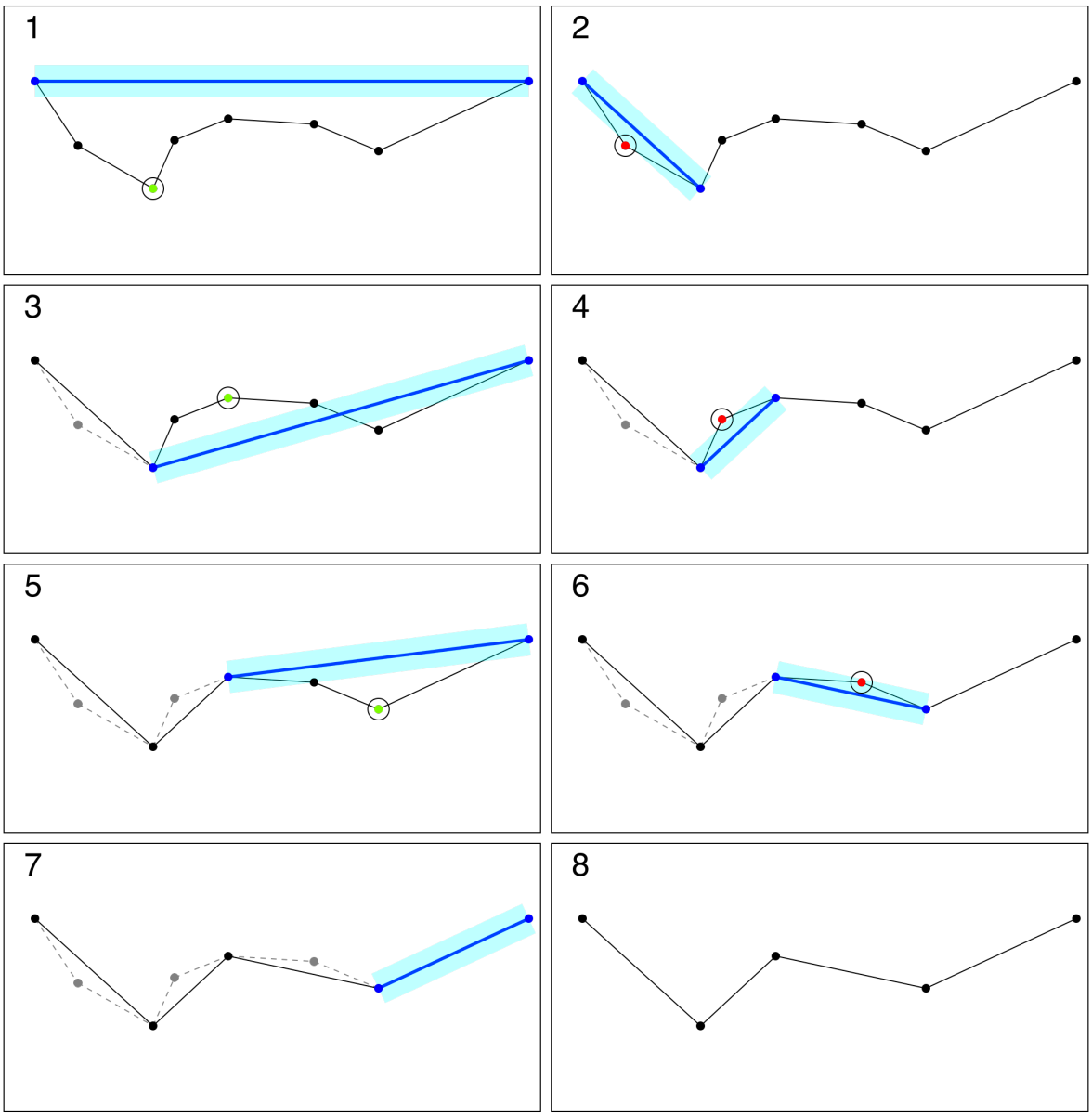


Figure 3.5: An example of the Ramer-Douglas-Peucker algorithm. The cyan area demarcates maximum distance values smaller than ϵ. Each step evaluates the point furthest away from the line between the current points. Points marked red are removed, whereas points marked green remain. Dashed lines represent removed line segments.

- $\delta > \epsilon$: the furthest point must be kept. The algorithm recurses with the same start point and the furthest point as the end point.

- When two points are evaluated with no intermediate points, the last line segment has been reached and the algorithm terminates.

An application of the algorithm is shown in Figure 3.5, where the line in the last frame represents the simplified line. Pseudocode of its functionality is shown in Figure 3.6.

```
 1: function RDP(points, ϵ)
 2:     d_max ← 0
 3:     index ← 0
 4:     last ← length(points)−1
 5:     for i ← 1 to last do
 6:         d ← PERPENDICULARDISTANCE(points[i], points[0], points[last])
 7:         if d > d_max then
 8:             index ← i
 9:             d_max ← d
10:         end if
11:     end for
12:     if d_max > ϵ then
13:         left_results ← RDP(points[0..index], ϵ)
14:         right_results ← RDP(points[index..last], ϵ)
15:         final_results ← left_results + right_results
16:     else
17:         final_results ← {points[0], points[last]}
18:     end if
19:     return final_results
20: end function
```

Figure 3.6: Pseudocode for the Ramer-Douglas-Peucker algorithm

3.6 Polyline to cubic Bézier conversion

Finally, we apply an algorithm for fitting a set of points to a curve that was developed by Schneider in 1990 [32]. The algorithm works as follows, for error threshold ϵ:

- Compute the approximate tangents at the start and end points of the curve by fitting a least-squares line to the points in the neighbourhood of the start and end point. These tangents provide the direction of the second and third control points.

- Assign an initial parameter value u_i to each point d_i using chord-length parameterisation.

- Calculate distances α_1 and α_2 for the second and third control points.

- Fit a cubic Bézier curve with control points $\{p_0, p_1, p_2, p_3\}$ to the input points. p_0 is equal to the start point, p_1 is equal to $\alpha_1 *$ left tangent, p_2 is equal to $\alpha_2 *$ right tangent and p_3 is equal to the end point.

- Compute the error η between the generated curve and input points. Two cases can be distinguished:

  - $\eta > \epsilon$: even though the error is unacceptable, chord-length parameterisation is a fast but inaccurate method of approximating the points. By applying Newton-Raphson iteration, the points are approximated more accurately. We try to fit the new values to a single cubic Bézier curve once more. If the error is still too large, we split the input data at the point of greatest error and recursively try and fit a curve to each data set.

- $\eta < \epsilon$: the input data can be fitted to a curve with an acceptable error margin, and we return the control points of the curve.

- When the last recursive call terminates, the algorithm returns a set of control points for each curve segment that is needed to draw the input data points.

Along with the chapter in which the algorithm is described, Schneider published C code that implements the algorithm. Pseudocode of the algorithm is shown in Figure 3.7.

```
 1: function FitCurve(points, ε)
 2:     last ← length(points) −1
 3:     left_tangent ← ComputeUnitTangent(points[0], points[1])
 4:     right_tangent ← ComputeUnitTangent(points[last-1], points[last])
 5:     FitCubic(points, left_tangent, right_tangent, ε)
 6: end function
 7: function FitCubic(points, left_tangent, right_tangent, ε)
 8:     parameterised_points ← ComputeChordLengthParameterisation(points)
 9:     bezier_curve ← ConstructBezier(left_tangent, right_tangent, parameterised_points)
10:     maximum_error ← ComputeMaxError(bezier_curve, points)
11:     if maximum_error < ε then
12:         return bezier_curve
13:     end if
14:     if maximum_error < ε² then
15:         for i ← 0 to max_iterations do
16:             parameterised_points ← ReparameteriseNewtonRaphson(points)
17:             bezier_curve ← ConstructBezier(left_tangent, right_tangent, parameterised_points)
18:             maximum_error, split_point ← ComputeMaxError(bezier_curve, points)
19:             if maximum_error < ε then
20:                 return bezier_curve
21:             end if
22:         end for
23:     end if
24:     ▷ Fitting failed, retry with split elements
25:     center_tangent ← ComputeUnitTangent(points[split_point-1], points[split_point+1])
26:     left_beziers ← FitCubic(points[0..split_point], left_tangent, center_tangent, ε)
27:     right_beziers ← FitCubic(points[split_point..last], center_tangent, right_tangent, ε)
28:     return left_beziers + right_beziers
29: end function
```

Figure 3.7: Pseudocode for fitting a cubic Bézier curve to a set of points

3.7 SVGO

SVGO is a tool written with Node.js that performs a number of optimisations on an SVG file without changing the image visually [35]. Examples of these optimisations are

- removing comments,
- removing hidden elements,
- collapsing nested groups,
- trimming whitespace,
- reducing decimal precision.

In some cases, SVGO reduces the file size up to 90%, although this only applies to very poorly structured SVG files. Generally, a reduction between 20% and 60% is expected.

3.8 Gzip

The gzip compression tool is based on the DEFLATE algorithm. This algorithm combines LZ77 compression with Huffman encoding for lossless compression [1].

LZ77 compression works by replacing words that have occurred before by replacing the word with a reference to the first occurrence. The reference is composed with a length and offset parameter in bytes. When the compressed string is decoded, the references are replaced with the words again.

Huffman encoding is then applied in order to compress the symbols that were unaffected by LZ77 compression. In essence, Huffman encoding determines the frequency at which individual characters occur. It then encodes the characters in bytes, where characters that occur the most frequent are encoded in the smallest byte representation. While the least frequent used characters might consume more space in their encoded form, on average this technique will reduce file size.

A downside of gzipping SVG files is that it no longer allows the file to be rendered progressively [44]. Most SVG viewers, such as those of modern browsers, have implemented their render engine in such a way that the SVG file is rendered even if it is not fully loaded. This is particularly convenient for larger files, since it allows the user to interact with SVG elements as soon as they are loaded.

3.9 Software stack

We have opted to use Python [37] for our prototype implementation. The main advantages of Python are its simplicity and its repository with a large amount of third party libraries. Python is considerably slower than C++ [21], the language used by Voormedia for their existing compression tools. However, since performance in terms of speed is not within the scope of this research, this is not considered to be of issue.

SVG processing The prototype uses two libraries for processing SVG files. Svgpathtools [27] is a library that offers an object-oriented model for SVG elements. Treating elements as objects simplifies the processing done on them, as element attributes are stored as object variables. Furthermore, path manipulation is intuitive as it is an extension of Python's mutable sequence. However, we found that svgpathtools lacked support for some SVG components. Filters and gradients were the most critical, as they were ignored and therefore not reproduced in the output file. As a consequence, important visual details were lost. To overcome this issue, we incorporated a second library called pysvg [20]. This library is similar to traditional XML parsers, and employs more rudimental methods of storing element attributes. However, it supports virtually all SVG specifics and thus produces a correct output file. We modified the parser module of pysvg in such a way that it initialises corresponding shape objects from svgpathtools. The shape objects are inserted in their respective parse tree nodes.

Mathematical operations The implementation of the steps described in Sections 3.3 to 3.5 are implemented with NumPy [8]. Among other things, NumPy adds support for multi-dimensional arrays and matrices, and a range of widely used calculations on these data types.

Polyline to cubic Bézier conversion As mentioned in Section 3.6, Schneider published an implementation of his algorithm in C [31]. This code snippet has been ported to Python and published on GitHub [26]. We altered the Python implementation in order for it to seamlessly integrate with the libraries described in this section.

Chapter 4

Experiment design

This chapter describes the experiments that are conducted to evaluate results produced by the prototype. Section 4.1 describes the goals of the experiment. Section 4.2 explains how image fidelity is validated, and Section 4.3 details how difference in file size is measured. The dataset is described in Section 4.4. Finally, Section 4.5 outlines the implementation of the experiment.

4.1 Experiment goals

By conducting this experiment, we aim to determine in which areas our proposed method of compression performs adequately and in which it does not. As the proposed method emphasises curves in SVG images, intuitively we can expect that images containing a large amount of curves produce better results compared to those with a scarce amount of curves. The results of the experiment should expose the optimal parameter values for the steps described in Section 3.5 Section 3.6. Furthermore, the results will clarify what elements contribute to a positive result, and problematic elements be identified for further research.

4.2 Image fidelity validation

We validate the images that are produced by the prototype based on the fidelity of the output file. Since the applied compression is lossy, it is expected that some loss of fidelity occurs. By measuring the difference in fidelity between the input and output file, we obtain the first half of the results needed for evaluating our proposed solution.

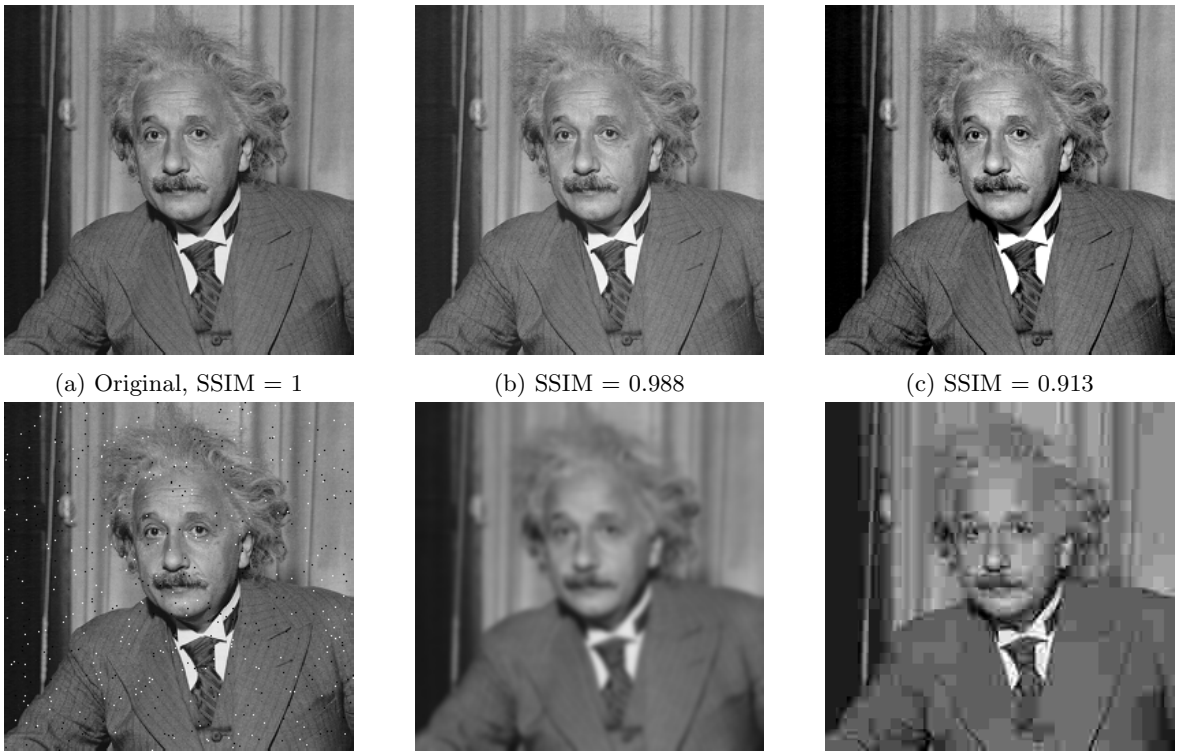
4.2.1 Image comparison

To the best of our knowledge, there is no robust method of directly comparing two SVG images. While a promising metric to achieve 'out-of-the-box' SVG image comparison has been proposed [18], it is incomplete and further development seems to have come to a halt. A method has been developed for comparing SVG files based on semantic features to facilitate content based image retrieval from a database [11]. However, isolating the comparison arithmetics and implementing them is too complex to be applied within the scope of this research. Thus, considering that image comparison is only required for our experiment, we have opted to take an alternative approach.

To determine the fidelity difference between the input and output files, the structural similarity (SSIM) index [48] is calculated. This metric is used by Voormedia in the regression tests for their existing compression tools. Since its publication the SSIM index has risen to be one of the most popular methods for image comparison, with the original paper currently boasting over 15,000 citations.

The SSIM index is calculated by comparing two images with respect to three aspects that relate to human perception of images [17]:

- **Luminance variation** compares the brightness of the images.



(a) Original, SSIM = 1 (b) SSIM = 0.988 (c) SSIM = 0.913

(d) SSIM = 0.840 (e) SSIM = 0.694 (f) SSIM = 0.662

Figure 4.1: SSIM index examples, where the value represents the comparison of each figure with Figure 4.1a. These images are part of the data set used by Wang in his research [47]

- **Contrast variation** compares the difference in range of darkest to brightest values of the images.
- **Correlation** compares the structure of the images by measuring the covariance between the images.

The algorithm works by scaling the images so that the resolutions are identical. A window with fixed dimensions is picked that evaluates a segment of the images for the three aspects described above. This produces a single score for the segment. After the last segment is evaluated, the individual scores are averaged. The result is a value in the range $[0.0, 1.0]$, where 0 denotes no correlation whatsoever and 1 indicates identical images. An example of measured SSIM index values is shown in Figure 4.1.

4.2.2 Acceptable loss of fidelity

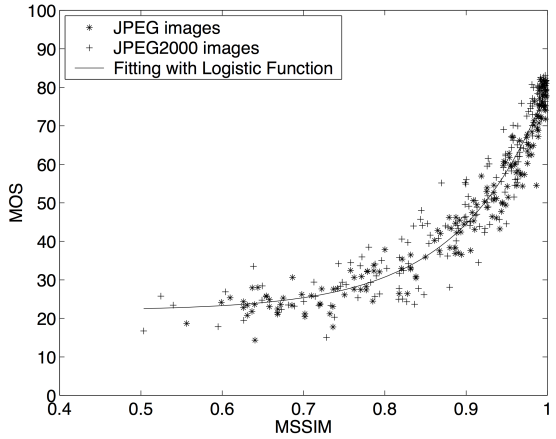
While the SSIM index is useful to determine differences between two images, we cannot give the measurements any value without a threshold value for what is deemed acceptable in terms of fidelity loss. In the paper that introduced the metric, Wang et al. validated their solution by comparing subjective results with objective results produced by SSIM index calculation [48].

The subjective results were obtained by letting a subject group rank an image dataset consisting of 344 images that resulted from compressing 29 base images with varying coefficients. The compression results were ranked 'bad', 'poor', 'fair', 'good', or 'excellent'. After processing the raw data, mean opinion scores (MOSs) were calculated for each image.

By plotting the MOSs against the SSIM index values that were measured for the same images, it became clear that the SSIM index performs well in objectively assessing image fidelity. Figure 4.2 shows the plot as reported in the original paper.

Zinner et al. [50] used the results reported by Wang et al. to develop a mapping between subjective and objective fidelity perception. Figure 4.3 shows the corresponding range of SSIM index values

for each MOS. Following this mapping, we are interested in measuring SSIM index values of 0.95 or greater, as this maps to a MOS of 4 (good) or 5 (excellent).



MOS	SSIM
5 (excellent)	≥ 0.99
4 (good)	≥ 0.95 & < 0.99
3 (fair)	≥ 0.88 & < 0.95
2 (poor)	≥ 0.5 & < 0.88
1 (bad)	< 0.5

Figure 4.3: Mapping of MOS to SSIM

4.3 File size reduction

The second factor of successful compression is self-explanatory. The output file needs to be smaller in size than the input file. An equal or even larger file size after compression is unjustifiable. During the experiment, the input file will be processed by the entire pipeline as described in Section 3.2. To ensure that we are only measuring the result of the lossy compression step, the input file is also processed with SVGO and gzip. The file size of the output file is then compared to the input file. The result is expressed in percentages.

4.4 Dataset

The dataset is composed of a total of 53 SVG images, spread over four different categories:

- **Clipart (13 images):** these types of images usually have more complex shapes, which we expect to deliver the best results.
- **Logos without text (16 images):** while logos may also contain complex shapes, they are part of a brand identity. The loss of fidelity might thus be less lenient as they are carefully designed to look like they do.
- **Logos with text (15 images):** the same applies here, with the additional constraint of embedded text. Text in logos is usually displayed as a path instead of a text element to correctly display custom fonts.
- **Images with embedded image (9 images):** raster images can be embedded in SVG images by either providing a URL to an external image, or by encoding the image in Base64. The encoded string can then be supplied to the SVG `<image>` tag as a data URI, which makes the SVG file self-contained. Depending on the complexity of the embedded image, the encoded string can be very long. Since the string must remain unaltered in the output file, the string length might contribute a considerable amount to the file size. In this case the minor file size reduction would no longer weigh up to the loss in fidelity.

The clipart images and images with embedded raster images were collected with the API provided by Openclipart[1]. Images from the base set were parsed and filtered on `<image>` tags with an encoded

[1] `https://openclipart.org/`

string to form the latter category. The logos were collected from Voormedia's image repository.

4.5 Experiment implementation

Similar to the prototype, the experiment is also implemented in Python. Python provides the libraries we need for evaluating the results, and is an excellent tool for performing analyses due to its simple nature. It also allows us to directly access intermediate results during execution.

4.5.1 SSIM index value measurement

Since the SSIM index value performs raster-based calculations, it can not be calculated using SVG images as input. Hence, the input and output files are first converted to PNG images with Cairo [49], a popular open source library for vector graphics manipulation.

The PNG images are loaded with PIL's Image module [19], which returns a numerical representation of the loaded image. The SSIM index values are then calculated using the implementation of scikit-image [36]. The function `ssim` is found in the scikit-image's measure module and takes images `X` and `Y` as input.

4.5.2 Parameter bounds

We determined sensible lower and upper bounds for the parameters of each step discussed in Sections 3.4 to 3.6 by trial and error. It became apparent that the number of iterations for De Casteljau's algorithm did not have to be tested for a range of values. Since the Ramer-Douglas-Peucker algorithm simplifies the polylines in the next step, SSIM index values did not significantly improve beyond 200 iterations for De Casteljau's algorithm. Conversely, a lower amount of iterations impacted measured SSIM index values too drastically. This is due to the fact that the data piped to the Ramer-Douglas-Peucker algorithm is too coarse.

With respect to the Ramer-Douglas-Peucker algorithm, we set the testing range for the error threshold ϵ_{rdp} to $0.1 \leq \epsilon_{rdp} \leq 1.0$ with increments of 0.1. Smaller increments did not affect the measured SSIM index values enough, and $\epsilon_{rdp} > 1.0$ suffered from too much loss of fidelity.

Finally, the error threshold ϵ_{p2b} for converting polylines to cubic Bézier curves is set to a range of $1 \leq \epsilon_{p2b} \leq 4$ with increments of 1. The same reasoning as for ϵ_{rdp} applies to ϵ_{p2b}.

4.5.3 Execution

For each permutation $P(\epsilon_{rdp}, \epsilon_{p2b})$, a function `run(epsilon_rdp, epsilon_p2b)` is called. The experiment script walks through directories containing the dataset and inputs each file in the prototype with the provided error thresholds. Next, the difference in file size is calculated with Python's `os.path` module. The input and output files are converted to PNG and the SSIM index values are calculated. A diff image is also produced. Although it serves no purpose in this research, the execution time is measured with Python's `time.time` module for the sake of completeness. The results of each permutation are saved to a CSV file with the name `<category>result_table_dpe<`ϵ_{rdp}`>_se<`ϵ_{p2b}`>.csv`.

Chapter 5

Results

This chapter explains the method of presenting the results in Section 5.1. The results of the clipart images are reported in Section 5.2. Section 5.3 and Section 5.4 present the results of the logos without and with text respectively. Finally, Section 5.5 reports the results of embedded images.

5.1 Method of presentation

To be able to interpret the raw data produced by the experiment, the data is plotted as a number of scatter plots. The data is grouped by category. Each group is then subgrouped by the error margin ϵ_{p2b}. Following the bounds for ϵ_{p2b} as defined in Section 4.5.2, this results in four plots per category. Each plot shows the measured file size differences in percentages on the x-axis, and the measured SSIM index values on the y-axis. Each file from the dataset is plotted with a unique color. The results for each error margin ϵ_{rdp} as defined in Section 4.5.2 are plotted with unique markers in the color of the specific file. The SSIM index values as defined in Section 4.2.2 corresponding to 'good' (≥ 0.95) and 'excellent' (≥ 0.99) are plotted with a dashed and dotted line respectively. A vertical dashed line is plotted at 0% for easier reading.

The results are categorised as follows, for SSIM index value $SSIM$ and file size difference δ_{fs}:

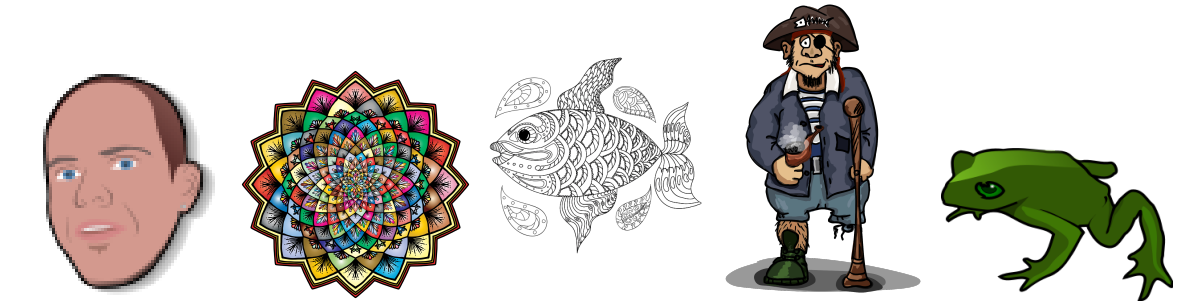
- **'Excellent' results:** $SSIM \geq 0.99 \wedge \delta_{fs} > 0\%$
- **'Good' results:** $0.95 \leq SSIM < 0.99 \wedge \delta_{fs} > 0\%$
- **'Failed' results:** $SSIM < 0.95 \vee \delta_{fs} < 0\%$

This section will include diff images which highlight differences between an input and output file. These differences are marked in pink. A greater difference between the two files is indicated by a larger surface of pink highlighting.

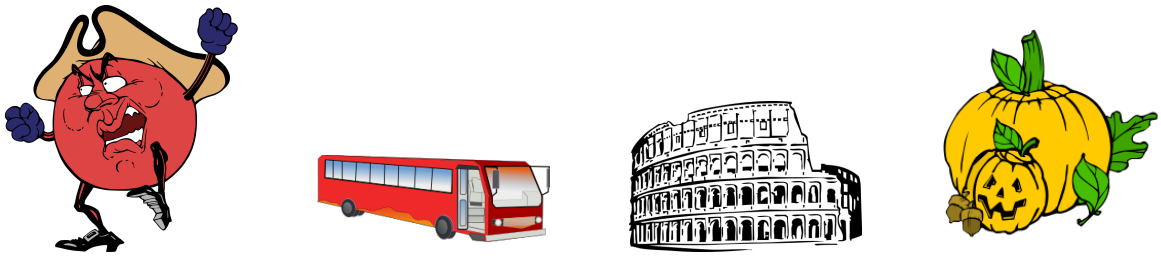
5.2 Clipart images

All cases in this category can be seen in Figure 5.1. The attentive reader will notice that case 7 is missing. This case failed to properly convert to PNG and was thus discarded from the dataset. Figures 5.12 and 5.13 show the results for the clipart images. The permutations of $(\epsilon_{rdp}, \epsilon_{p2b})$ per category are listed in Figure 5.11. The figures reveal a number of interesting cases per category to evaluate:

- **'Excellent' results:** case 1 and 6
- **'Good' results:** case 9 and 10, as they show the widest spread; case 1 and 6, to compare with the 'excellent' results; case 5, as the results are highly dense
- **'Bad' results:** case 14, as the results are virtually identical regardless of the permutation; case 11, as it shows the worst results



(a) case 11 (b) case 2 (c) case 3 (d) case 4 (e) case 13



(f) case 6 (g) case 14 (h) case 9 (i) case 10

(j) case 1 (k) case 12 (l) case 5 (m) case 8

Figure 5.1: Dataset images in the category ‘clipart’

5.2.1 'Excellent' results

Case 1 has three permutations that are part of this category, whereas case 6 only has one. For case 6, we evaluate the permutation with the highest compression rate. Figures 5.2 and 5.3 show the original, compressed and diff image. The result aligns with the classification 'excellent', as it is hard to spot any differences with the naked eye. However, the compression rates are small. Case 1 and case 6 have been compressed by 6.78% and 2.5% respectively.

(a) Original image (b) Compressed image (c) Diff image

Figure 5.2: Result of compressing case 1 for $\epsilon_{rdp} = 0.3, \epsilon_{p2b} = 1$ with $\delta_{fs} = 6.78\%$

(a) Original image (b) Compressed image (c) Diff image

Figure 5.3: Result of compressing case 6 for $\epsilon_{rdp} = 0.4, \epsilon_{p2b} = 1$ with $\delta_{fs} = 2.5\%$

5.2.2 'Good' results

We first report on case 1 and 6, to see how they compare to the 'excellent' results. Both files are evaluated for the same ϵ_{rdp} as in the 'excellent' result, but with $\epsilon_{p2b} = 4$. Case 1 is shown in Figure 5.4. Figure 5.4c shows that there is significantly more loss of fidelity, such as in the area where the hand goes in the pocket. The file size has been reduced by 27.3% for this permutation. As Figure 5.5 shows, case 6 does not seem to suffer as much from the loss in fidelity. The measured difference is mostly due to the variation in line thickness. Case 6 is compressed by 21.4% for this permutation.

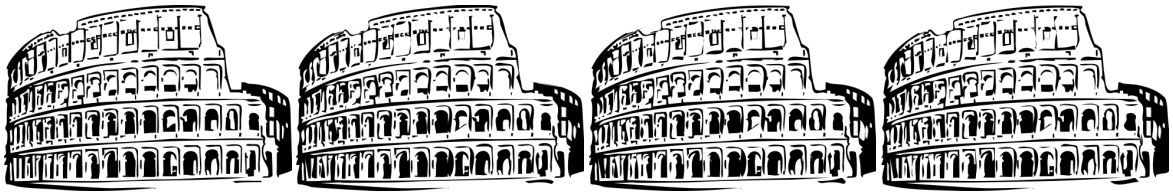
Case 9 and 10 are displayed in Figures 5.6 and 5.7 respectively. From the full range of permutations that fall in the category 'good', we show the lowest, middle, and highest values for ϵ_{rdp} and $\epsilon_{p2b} = 1$. Upon close observation the degradaton of fidelity is noticeable. Especially fine lines tend to become more coarse. However, when glanced over it is difficult to spot differences between the images. In these cases, the file size reduction is more or less linear.

(a) Original image (b) Compressed image (c) Diff image

Figure 5.4: Result of compressing case 1 for $\epsilon_{rdp} = 0.3, \epsilon_{p2b} = 4$ with $d_{fs} = 27.3\%$

(a) Original image (b) Compressed image (c) Diff image

Figure 5.5: Result of compressing case 6 for $\epsilon_{rdp} = 0.4, \epsilon_{p2b} = 4$ with $d_{fs} = 21.4\%$



(a) Original image (b) $\epsilon_{rdp} = 0.3, \delta_{fs} = 4.39\%$ (c) $\epsilon_{rdp} = 0.5, \delta_{fs} = 7.97\%$ (d) $\epsilon_{rdp} = 0.7, \delta_{fs} = 11.4\%$

Figure 5.6: Results for case 9 with lowest, middle and highest ϵ_{rdp} and $\epsilon_{p2b} = 1$ that fall in the category 'good'

(a) Original image (b) $\epsilon_{rdp} = 0.1, \delta_{fs} = 21.3\%$ (c) $\epsilon_{rdp} = 0.5, \delta_{fs} = 31\%$ (d) $\epsilon_{rdp} = 1.0, \delta_{fs} = 40.9\%$

Figure 5.7: Results for case 10 with lowest, middle and highest ϵ_{rdp} and $\epsilon_{p2b} = 1$ that fall in the category 'good'

Case 5 is seen in Figure 5.8. Since the SSIM index values for this case are so densely clustered, we picked the highest value for ϵ_{rdp} for each value of ϵ_{p2b}. As the images show, it is very hard to spot differences between the images. The vast amount of lines in this case make it hard for the human eye to spot differences. However, these lines have low rates of compression: the most lenient setting only achieves a 7.09% compression result.

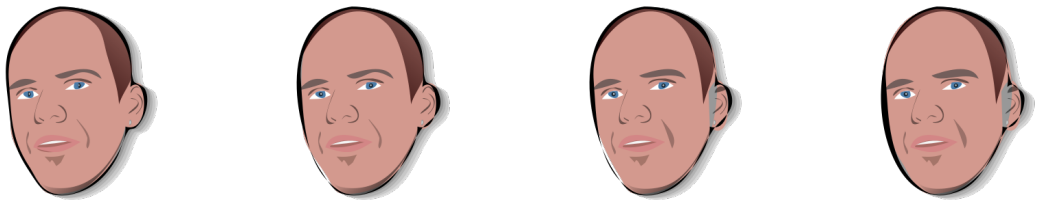
(a) $\epsilon_{p2b} = 1, \delta_{fs} = 1.93\%$ (b) $\epsilon_{p2b} = 2, \delta_{fs} = 3.82\%$ (c) $\epsilon_{p2b} = 3, \delta_{fs} = 5.22\%$ (d) $\epsilon_{p2b} = 4, \delta_{fs} = 7.09\%$

Figure 5.8: Results for case 5 with $\epsilon_{rdp} = 1.0$

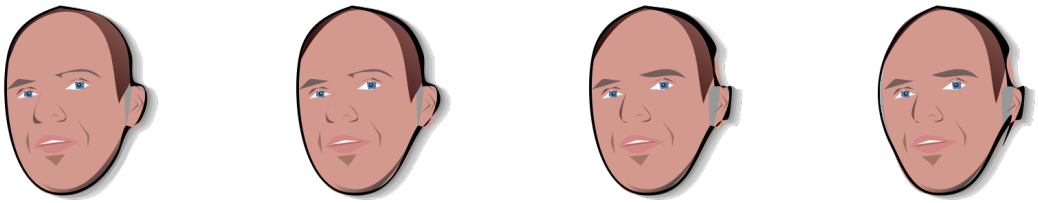
5.2.3 'Bad' results

Figures 5.9 and 5.10 show the results for case 11 and 14 respectively. Case 14 has an issue with the top left corner of the bus windows. Varying parameter permutations have virtually no impact on the file size difference, which does not deviate far from 2%.

The subfigures in Figure 5.10 show the lowest and highest values for ϵ_{rdp} on the top and bottom rows respectively. Here we can see that both the line simplification as well as the curve fitting drastically impacts the fidelity of the images. While the file size reductions vary between 33–48%, the images are heavily distorted.

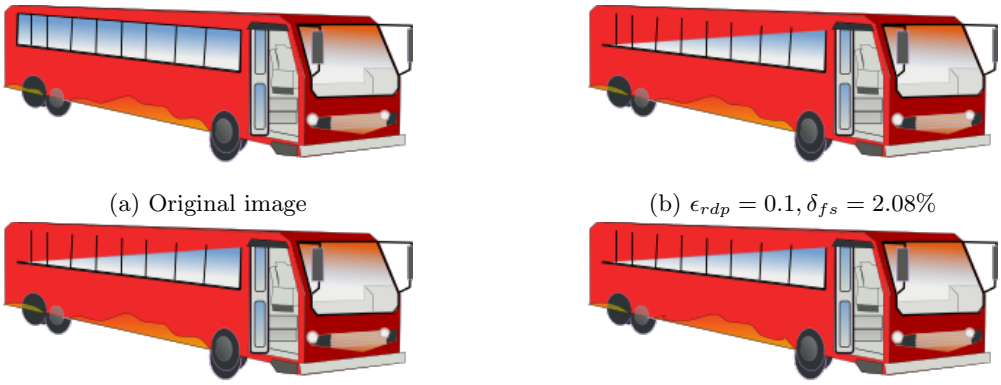


(a) $\epsilon_{rdp} = 0.1, \epsilon_{p2b} = 1$ (b) $\epsilon_{rdp} = 0.1, \epsilon_{p2b} = 2$ (c) $\epsilon_{rdp} = 0.1, \epsilon_{p2b} = 3$ (d) $\epsilon_{rdp} = 0.1, \epsilon_{p2b} = 4$



(e) $\epsilon_{rdp} = 1.0, \epsilon_{p2b} = 1$ (f) $\epsilon_{rdp} = 1.0, \epsilon_{p2b} = 2$ (g) $\epsilon_{rdp} = 1.0, \epsilon_{p2b} = 3$ (h) $\epsilon_{rdp} = 1.0, \epsilon_{p2b} = 4$

Figure 5.9: Results for case 11



(a) Original image (b) $\epsilon_{rdp} = 0.1, \delta_{fs} = 2.08\%$

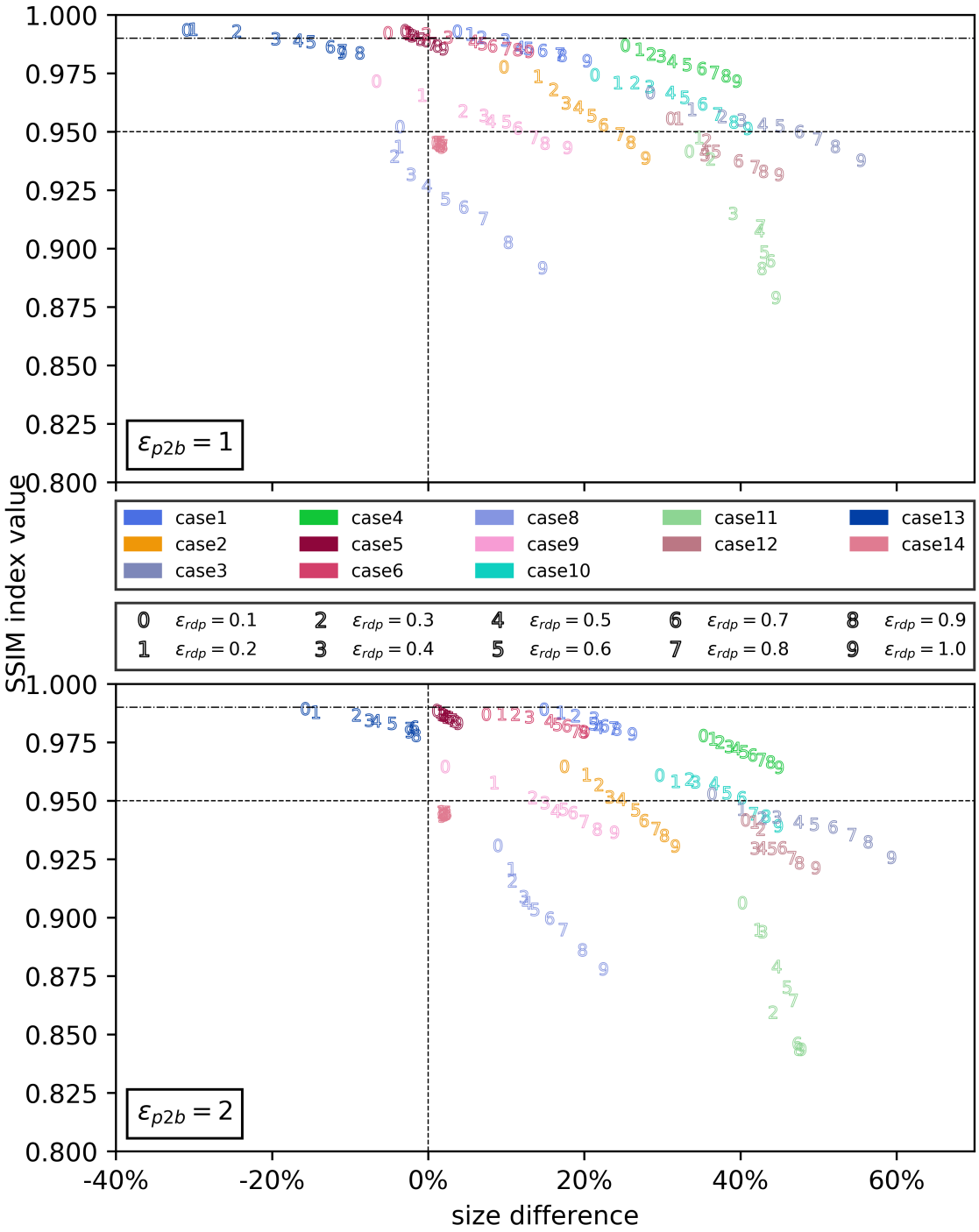
(c) $\epsilon_{rdp} = 0.5, \delta_{fs} = 2.28\%$ (d) $\epsilon_{rdp} = 1.0, \delta_{fs} = 2.35\%$

Figure 5.10: Results for case 14 with $\epsilon_{p2b} = 4$

	'Excellent' ϵ_{p2b} values	**'Good' ϵ_{p2b} values**			
	1	**1**	**2**	**3**	**4**
case1	{.1, .2, .3}	{.4, ..., 1.}	{.1, ..., 1.}	{.1, ..., 1.}	{.1, ..., 1.}
case2		{.1, ..., .7}	{.1, ..., .5}	{.1, .2, .3}	{.1, .2}
case3		{.1, ..., .7}	{.1}		
case4		{.1, ..., 1.}	{.1, ..., 1.}	{.1, ..., 1.}	{.1, ..., 1.}
case5		{.8, .9, 1.}	{.1, ..., 1.}	{.1, ..., 1.}	{.1, ..., 1.}
case6	{.4}	{.5, ..., 1.}	{.1, ..., 1.}	{.1, ..., 1.}	{.1, ..., 1.}
case9		{.3, ..., .7}	{.1, .2, .3}	{.1, .2}	{.1, .2}
case10		{.1, ..., 1.}	{.1, ..., .7}	{.1, .2, .3}	{.1}
case12		{.1, .2}			
case13				{.8, 1.}	{.6, ..., 1.}

Figure 5.11: Permutations of $(\epsilon_{rdp}, \epsilon_{p2b})$ for clipart images in their respective categories. The cells contain the values for ϵ_{rdp}

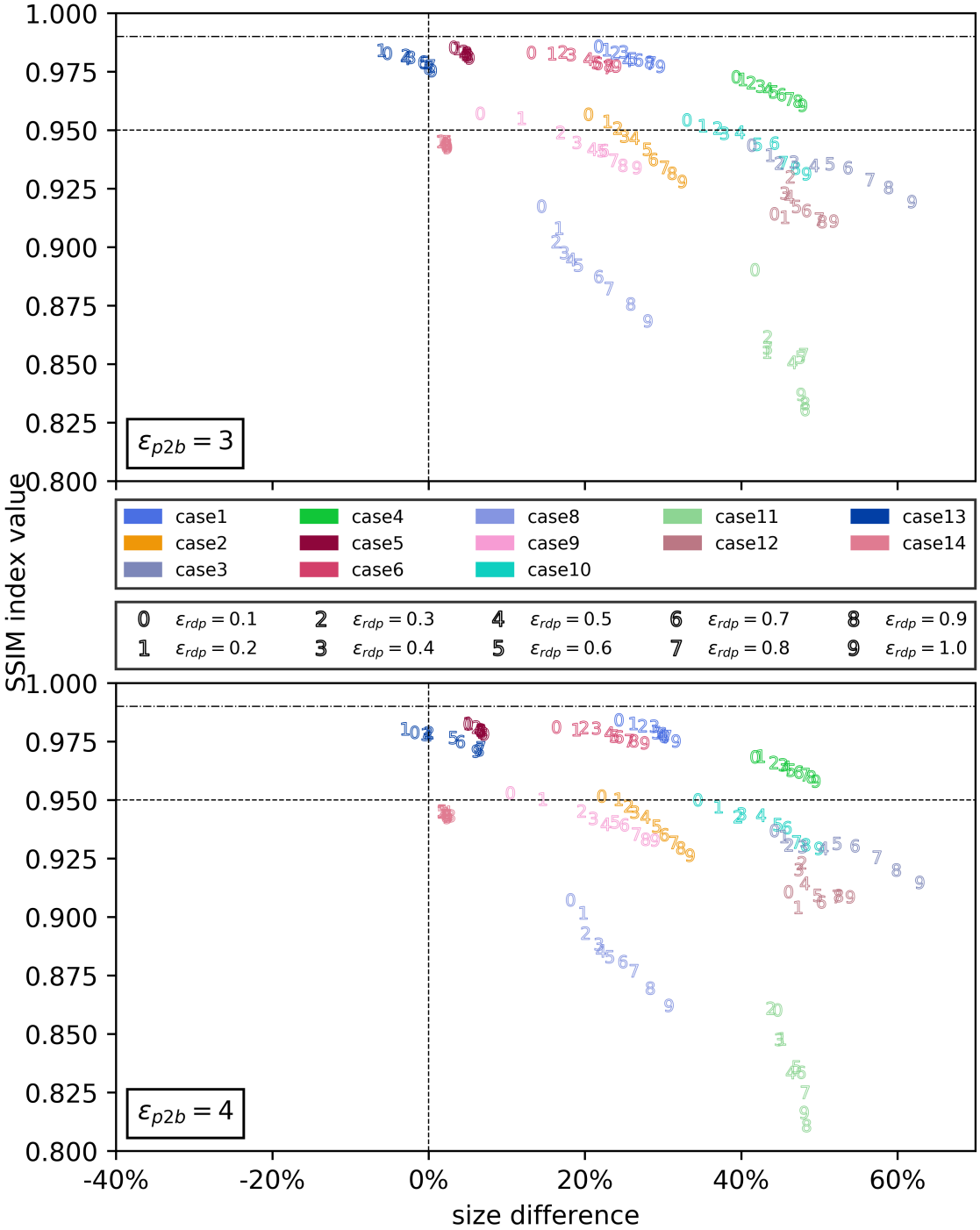
(a) Results for clipart images with $\epsilon_{p2b} = 1$



(b) Results for clipart images with $\epsilon_{p2b} = 2$

Figure 5.12: Results of clipart images for $\epsilon_{rdp} \in \{0.1, 0.2, 0.3, \cdots, 1.0\}$ and $\epsilon_{p2b} \in \{1, 2\}$

(a) Results for clipart images with $\epsilon_{p2b} = 3$



(b) Results for clipart images with $\epsilon_{p2b} = 4$

Figure 5.13: Results of clipart images for $\epsilon_{rdp} \in \{0.1, 0.2, 0.3, \cdots, 1.0\}$ and $\epsilon_{p2b} \in \{3, 4\}$

5.3 Logos without text

All cases in this category can be seen in Figure 5.14. Figures 5.23 and 5.24 show the results for logos without text. The permutations of $(\epsilon_{rdp}, \epsilon_{p2b})$ per category are listed in Figure 5.22. For this category, we evaluate the following results:

- **'Excellent' results:** case 4 and 14
- **'Good' results:** case 4 and 14, in order to compare with the 'excellent' results; case 10, as the results show little variation and all are qualified as 'good'; case 13, as lower ϵ_{rdp} values seem to produce higher compression results for $\epsilon_{p2b} = 2$
- **'Bad' results:** case 2 and 3, since they have a SSIM index value of 1 and produce identical results regardless of the permutation; case 9, as it performs the worst

(a) case 1 (b) case 2 (c) case 3 (d) case 4

(e) case 5 (f) case 6 (g) case 7 (h) case 8

(i) case 9 (j) case 10 (k) case 11 (l) case 12

(m) case 13 (n) case 15 (o) case 16 (p) case 14

Figure 5.14: Dataset images in the category 'logos without text'

5.3.1 'Excellent' results

Figures 5.15 and 5.16 show the results for case 4 and 14 respectively. Case 4 does a good job of compressing, with no clear loss of fidelity. The file size difference is significant with 32.9%. Case 14 also has significant compression results, although the loss in fidelity is more noticable. Notice the three holes in the shield; the top left hole is no longer square, and the other two are removed completely.

(a) Original image (b) Compressed image (c) Diff image

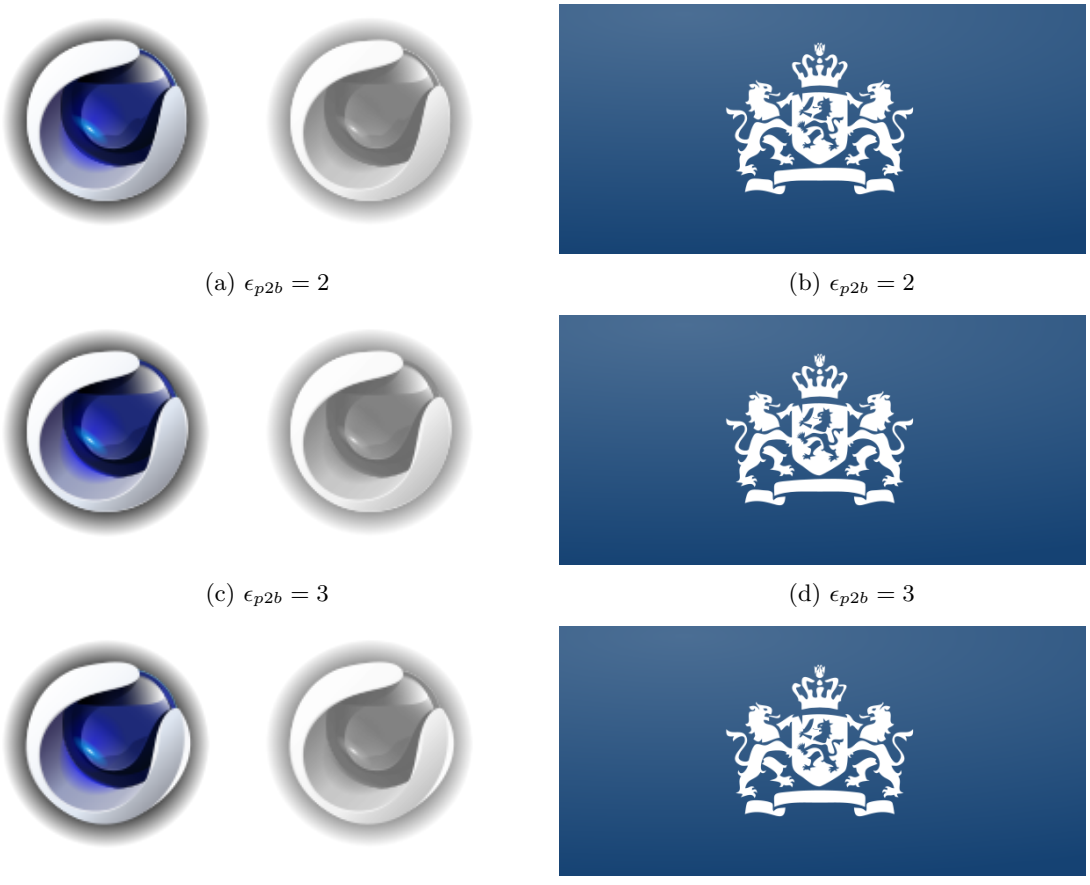
Figure 5.15: Result of compressing case 4 for $\epsilon_{rdp} = 0.4, \epsilon_{p2b} = 1$ with $\delta_{fs} = 32.9\%$

(a) Original image (b) Compressed image (c) Diff image

Figure 5.16: Result of compressing case 14 for $\epsilon_{rdp} = 0.4, \epsilon_{p2b} = 1$ with $\delta_{fs} = 30.1\%$

5.3.2 'Good' results

Figure 5.17 shows how varying ϵ_{p2b} values significantly impact the fidelity for both case 4 and 14. In case 4, the outer ring is no longer a circle which exposes the white surface behind it. The crown and the lion's legs in case 14 are either too thick or too thin.



(a) $\epsilon_{p2b} = 2$ (b) $\epsilon_{p2b} = 2$

(c) $\epsilon_{p2b} = 3$ (d) $\epsilon_{p2b} = 3$

(e) $\epsilon_{p2b} = 4$ (f) $\epsilon_{p2b} = 4$

Figure 5.17: Results of compressing case 4 and 14 for $\epsilon_{rdp} = 0.4$ and varying values for ϵ_{p2b}

Although all permutations of case 10 are reported as 'good', on closer inspection this case would be better classified as 'bad'. Figure 5.18 shows that for the strictest permutation, there are black lines crossing the yellow cube. For the most lenient permutation, the grip of the magnifying glass is a lot thicker than in the original, as well as suffering from the black lines.

(a) Original image (b) $\epsilon_{rdp} = 0.1, \epsilon_{p2b} = 1$ (c) $\epsilon_{rdp} = 1.0, \epsilon_{p2b} = 4$

Figure 5.18: Results for case 10 with various values for $\epsilon_{rdp}, \epsilon_{p2b}$

Figure 5.19 shows the results of compressing case 13 with the minimum and maximum value for ϵ_{rdp} and $\epsilon_{p2b} = 1$. Due to the simple nature, it is quite hard to discern the two. However, there is quite some difference between the file size reductions, where the stricter permutation performs better. This is unexpected, since less strict compression should result better results.

(a) Original image (b) $\epsilon_{rdp} = 0.1, \delta_{fs} = 71.3\%$ (c) $\epsilon_{rdp} = 1.0, \delta_{fs} = 64.2\%$

Figure 5.19: Results for case 13 with $\epsilon_{p2b} = 1$

5.3.3 'Bad' results

Figure 5.20 show the results of case 2 and 3. The SSIM index value of 1.0 for both images asserts that the images are identical before and after compression. It is easy to spot why: these logos are composed with straight lines only. Although no compression takes place, the file size is increased. This needs to be inspected.

(a) Original image (b) $\epsilon_{rdp} = 1.0, \epsilon_{p2b} = 4$ (c) Original image (d) $\epsilon_{rdp} = 1.0, \epsilon_{p2b} = 4$

Figure 5.20: Results for case 2 with $\delta_{fs} = -4.94\%$ and case 3 with $\delta_{fs} = -7.49\%$

Figure 5.21 displays case 9, the worst result of this category. The result is heavily distorted.

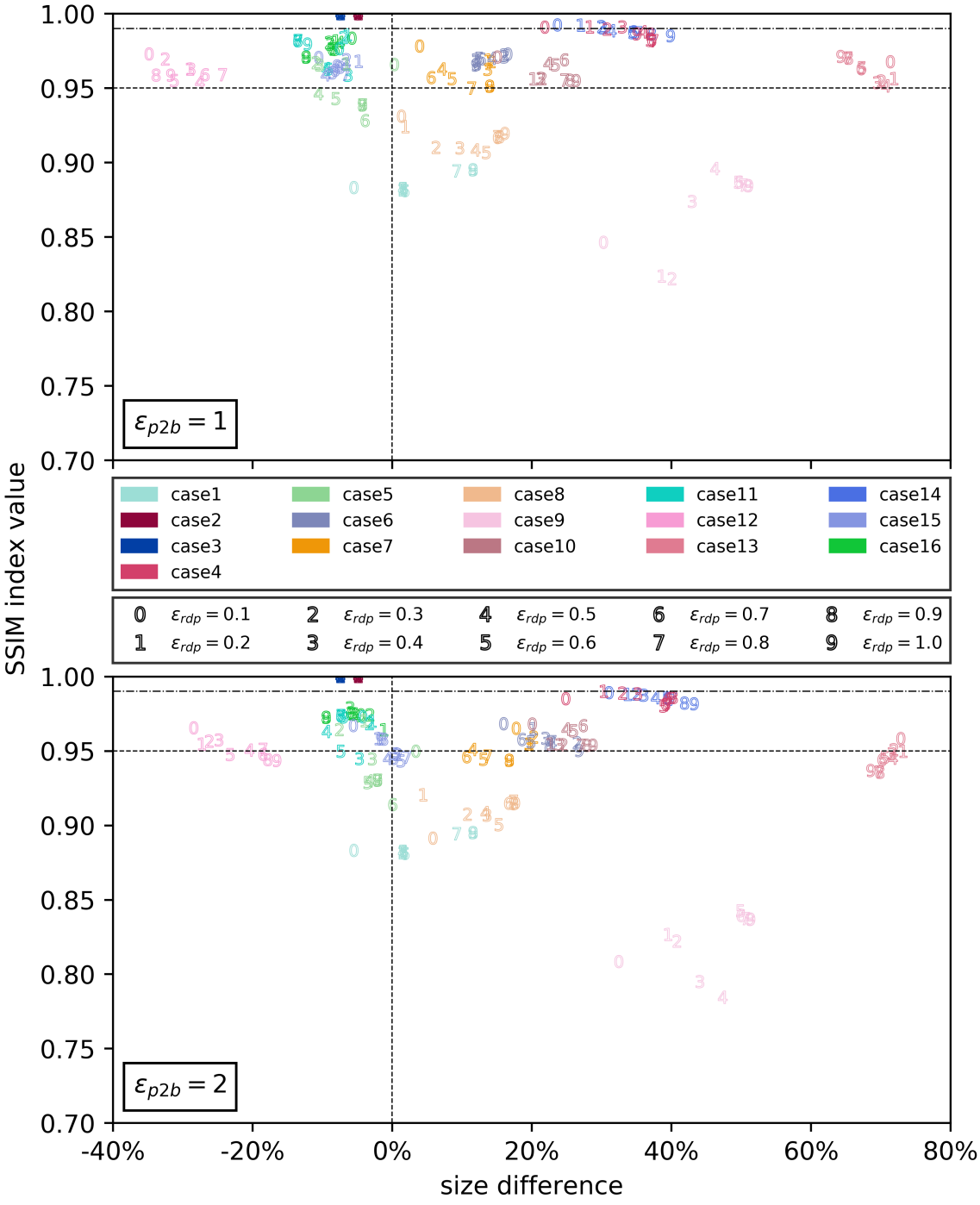
(a) Original image (b) Compressed image (c) Diff image

Figure 5.21: Result of compressing case 9 for $\epsilon_{rdp} = 0.5, \epsilon_{p2b} = 4$

	'Excellent' ϵ_{p2b} values		**'Good' ϵ_{p2b} values**			
	1	**2**	**1**	**2**	**3**	**4**
case4	$\{.1, .2, .4\}$	$\{.2\}$	$\{.3, .5, ..., 1.\}$	$\{.1, .3, ..., 1.\}$	$\{.1, ..., 1.\}$	$\{.1, ..., 1.\}$
case5			$\{.1\}$			
case6			$\{.1, ..., 1.\}$	$\{.1, ..., 1.\}$	$\{.1, ..., .8\}$	$\{.1, ..., .5, .8\}$
case7			$\{.1, ..., 1.\}$	$\{.1, ..., .5\}$	$\{.1, .2, .3\}$	$\{.1\}$
case10			$\{.1, ..., 1.\}$	$\{.1, ..., 1.\}$	$\{.1, ..., 1.\}$	$\{.1, ..., 1.\}$
case13			$\{.1, ..., 1.\}$	$\{.1, .3\}$	$\{.1\}$	$\{.1\}$
case14	$\{.1, ..., .4\}$		$\{.5, ..., 1.\}$	$\{.1, ..., 1.\}$	$\{.1, ..., 1.\}$	$\{.1, ..., 1.\}$
case16					$\{.5\}$	$\{.5\}$

Figure 5.22: Permutations of $(\epsilon_{rdp}, \epsilon_{p2b})$ for logos without text in their respective categories. The cells contain the values for ϵ_{rdp}

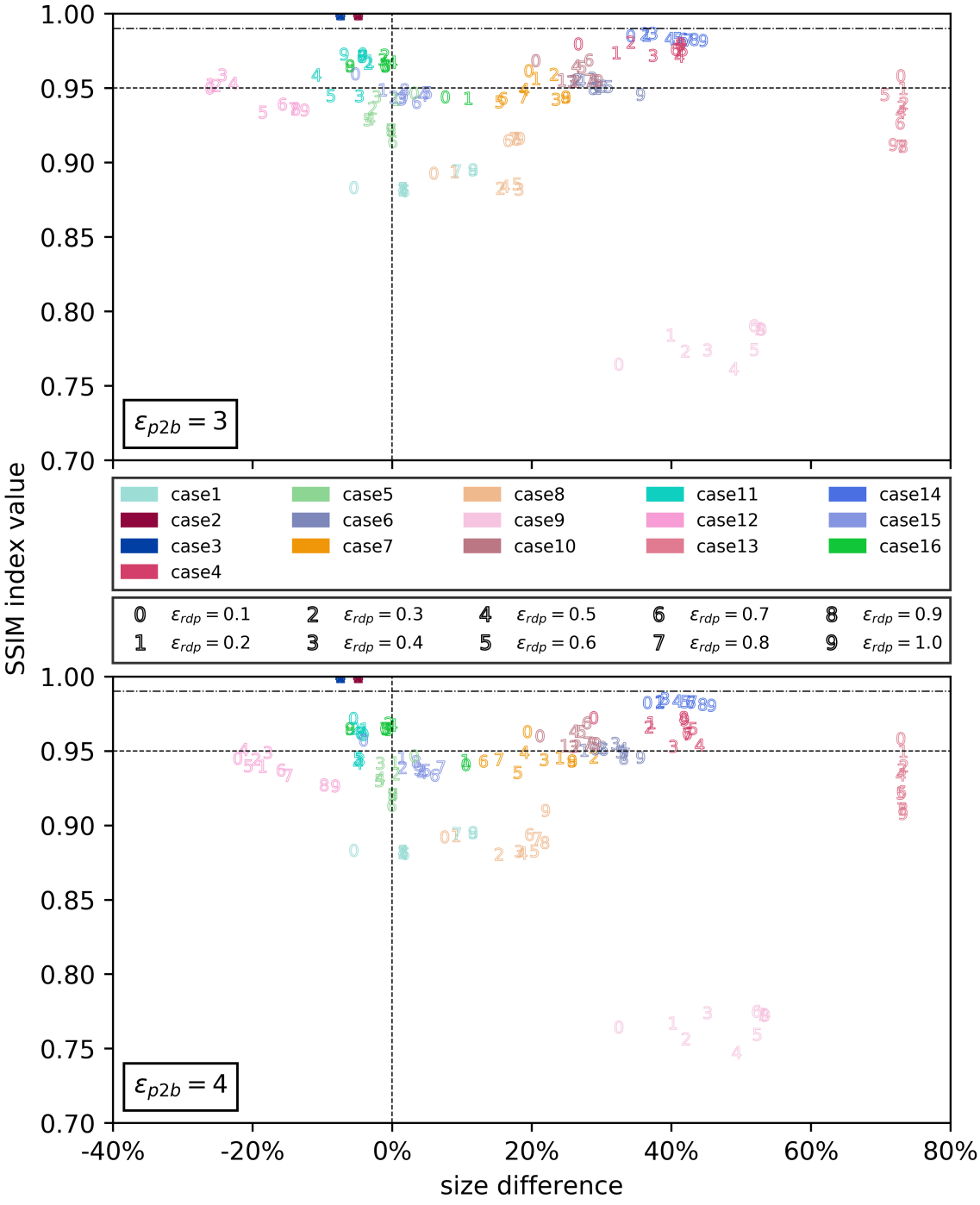
(a) Results for logos without text with $\epsilon_{p2b} = 1$



(b) Results for logos without text with $\epsilon_{p2b} = 2$

Figure 5.23: Results of logos without text for $\epsilon_{rdp} \in \{0.1, 0.2, 0.3, \cdots, 1.0\}$ and $\epsilon_{p2b} \in \{1, 2\}$

(a) Results for logos without text with $\epsilon_{p2b} = 3$



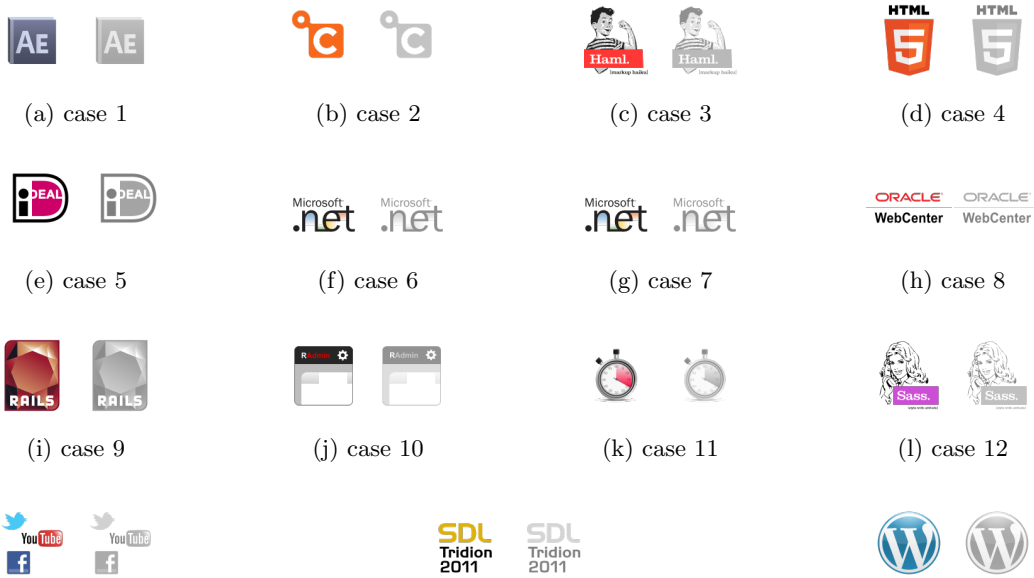
(b) Results for logos without text with $\epsilon_{p2b} = 4$

Figure 5.24: Results of logos without text for $\epsilon_{rdp} \in \{0.1, 0.2, 0.3, \cdots, 1.0\}$ and $\epsilon_{p2b} \in \{3, 4\}$

5.4 Logos with text

All cases in this category can be seen in Figure 5.25. Figures 5.34 and 5.35 show the results for logos with text. The permutations of $(\epsilon_{rdp}, \epsilon_{p2b})$ per category are listed in Figure 5.33. For this category, we evaluate the following results:

- **'Excellent' results:** case 11
- **'Good' results:** case 11, in order to compare with the 'excellent' results; case 12 and 15, as they show a very wide spread
- **'Bad' results:** case 1 and 4, since they have a SSIM index value of 1 and produce identical results regardless of the permutation; case 10, as it produces the worst results; case 6, 9 and 14 as a general performance review



(a) case 1 (b) case 2 (c) case 3 (d) case 4

(e) case 5 (f) case 6 (g) case 7 (h) case 8

(i) case 9 (j) case 10 (k) case 11 (l) case 12

(m) case 13 (n) case 14 (o) case 15

Figure 5.25: Dataset images in the category 'logos with text'

5.4.1 'Excellent' results

Figure 5.26 shows the result of the only case in the category 'excellent'. At the intended viewing size, it is difficult to see where the loss of fidelity occurs. However, upon zooming in it becomes clear that the clock numbers are suffering from the compression. Furthermore, the hand of the clock is slightly warped. In this specific case, these errors are not necessarily an issue. The logo intends to convey the concept of a stopwatch, which is not harmed by the compression. However, more lenient compression permutations make the numbers even harder to recognise without yielding significantly better compression results than the strictest permutation.

5.4.2 'Good' results

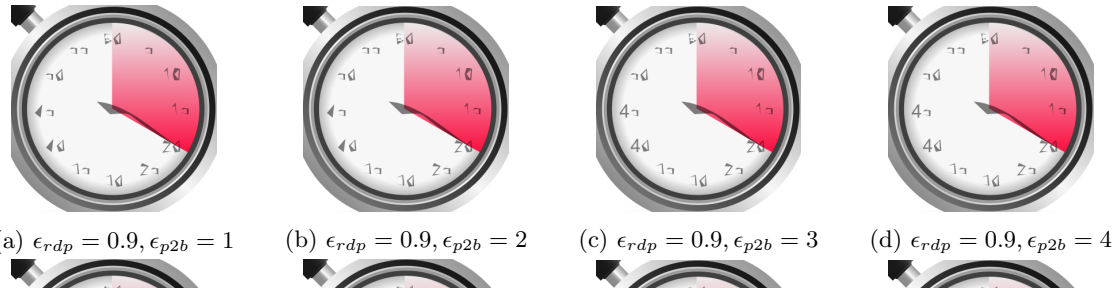
Case 11 is revisited for this category in Figure 5.27. A strange behaviour is observed for the highest values of ϵ_{p2b}: the number 4 is no longer compressed. The rest of the characters are mostly unrecognisable now, which aligns with the expectations.

Figures 5.28 and 5.29 show the results of case 12 and 15. For case 12 with strict compression permutations, the result is decent. This is likely due to the curved nature of the letters in the purple

(a) Original image (b) $\epsilon_{rdp} = 0.1, \delta_{fs} = 28.4\%$ (c) $\epsilon_{rdp} = 0.8, \delta_{fs} = 32.8\%$

(d) Closeup original (e) Closeup $\epsilon_{rdp} = 0.1$ (f) Closeup $\epsilon_{rdp} = 0.8$

Figure 5.26: Results for case 11 with $\epsilon_{p2b} = 1$



(a) $\epsilon_{rdp} = 0.9, \epsilon_{p2b} = 1$ (b) $\epsilon_{rdp} = 0.9, \epsilon_{p2b} = 2$ (c) $\epsilon_{rdp} = 0.9, \epsilon_{p2b} = 3$ (d) $\epsilon_{rdp} = 0.9, \epsilon_{p2b} = 4$

(e) $\epsilon_{rdp} = 1.0, \epsilon_{p2b} = 1$ (f) $\epsilon_{rdp} = 1.0, \epsilon_{p2b} = 2$ (g) $\epsilon_{rdp} = 1.0, \epsilon_{p2b} = 3$ (h) $\epsilon_{rdp} = 1.0, \epsilon_{p2b} = 4$

Figure 5.27: Closeups of the results for case 11 in the category 'good'

box. However, the subscript is already hard to read. With more lenient permutations, the letters become unpleasant to read, and the woman also becomes visibly less detailed.

(a) Original image

(b) $\epsilon_{rdp} = 0.1$

(c) $\epsilon_{rdp} = 0.5$

(d) $\epsilon_{rdp} = 1.0$

Figure 5.28: Results for case 12 with $\epsilon_{p2b} = 1$

Case 15 performs quite well for the strictest permutation. Although some small details are lost, this is not clearly visible. However, when ϵ_{rdp} is less strict, the letter quickly gets visibly distorted.

(a) Original image

(b) $\epsilon_{rdp} = 0.1$

(c) $\epsilon_{rdp} = 0.5$

(d) $\epsilon_{rdp} = 1.0$

Figure 5.29: Results for case 15 with $\epsilon_{p2b} = 1$

5.4.3 'Bad' results

Both case 1 and 4 remain unaltered after compression. As Figure 5.30 shows, these cases are comparable to the 'bad' results from Section 5.3. We need to investigate why the letters remain unchanged with these cases and why the file size increases.

Case 10 is shown in Figure 5.31. This case performed the poorest out of all cases in the experiment. There is no element in the image that was compressed correctly. The text is unreadable, straight lines are no longer straight, and curves are heavily distorted.

Case 6, 9 and 14 are shown in Figure 5.32. These images show that a solution for text compression is necessary in order for the algorithm to be applicable to a broad range of SVG images.

(a) Original image (b) $\epsilon_{rdp} = 1.0, \epsilon_{p2b} = 4$ (c) Original image (d) $\epsilon_{rdp} = 1.0, \epsilon_{p2b} = 4$

Figure 5.30: Results for case 1 with $\delta_{fs} = -3.09\%$ and case 4 with $\delta_{fs} = -3.75\%$

(a) Original image (b) Compressed image (c) Diff image

Figure 5.31: Result for case 10 with $\epsilon_{rdp} = 1.0, \epsilon_{p2b} = 4$

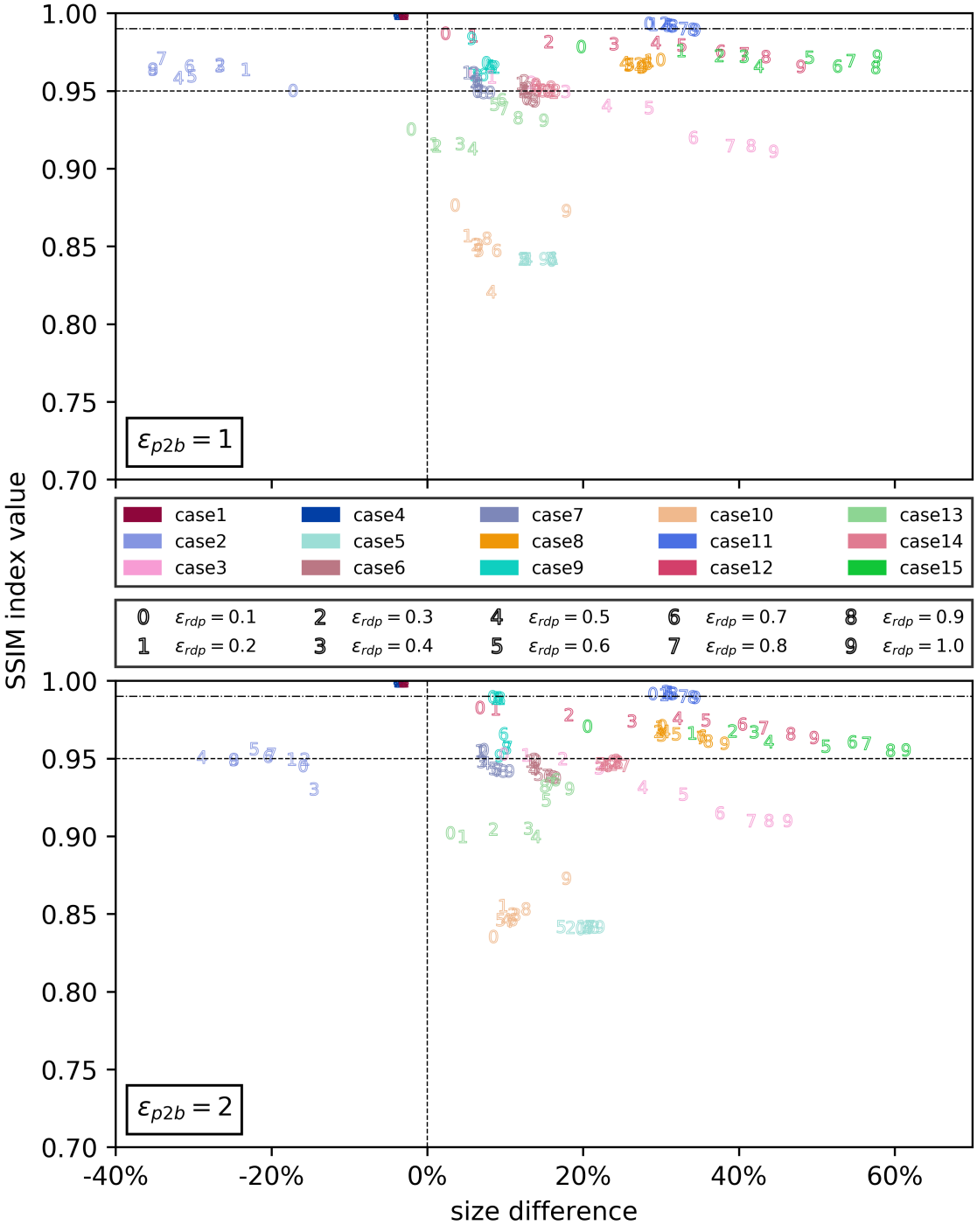
(a) Case 6 (b) Case 9 (c) Case 14

Figure 5.32: Results for $\epsilon_{rdp} = 0.1, \epsilon_{p2b} = 1$

	'Excellent' ϵ_{p2b} values	**'Good' ϵ_{p2b} values**			
	$\{1, ..., 4\}$	**1**	**2**	**3**	**4**
case3		$\{.1, .2, .3\}$	$\{.1, .2\}$		
case6		$\{.1, ..., .4\}$		$\{.3\}$	
case7		$\{.1, ..., .6\}$	$\{.1, .2, .3\}$	$\{.1, .2, .3\}$	$\{.1, .2, .3\}$
case8		$\{.1, ..., 1.\}$	$\{.1, ..., 1.\}$	$\{.1, ..., 1.\}$	$\{.1, ..., 1.\}$
case9		$\{.1, ..., 1.\}$	$\{.1, ..., 1.\}$	$\{.1, ..., .9\}$	$\{.1\}$
case11	$\{.1, ..., .8\}$	$\{.9, 1.\}$	$\{.9, 1.\}$	$\{.9, 1.\}$	$\{.9, 1.\}$
case12		$\{.1, ..., 1.\}$	$\{.1, ..., 1.\}$	$\{.1, ..., 1.\}$	$\{.1, ..., 1.\}$
case14		$\{.3, ..., 1.\}$		$\{1.\}$	$\{.9\}$
case15		$\{.1, ..., 1.\}$	$\{.1, ..., 1.\}$	$\{.1, ..., .8, 1.\}$	$\{.3, ..., .7\}$

Figure 5.33: Permutations of $(\epsilon_{rdp}, \epsilon_{p2b})$ for logos with text in their respective categories. The cells contain the values for ϵ_{rdp}

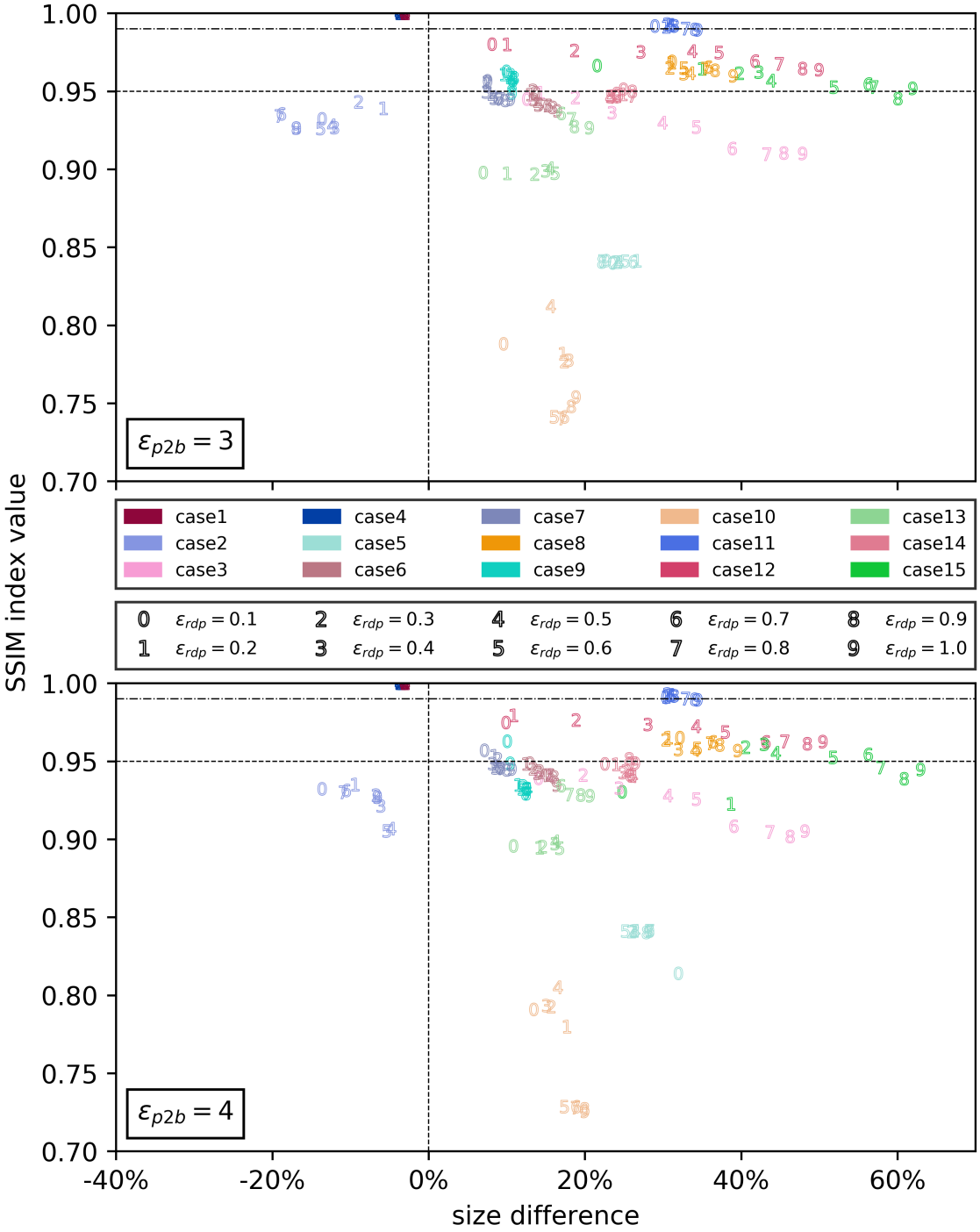
(a) Results for logos with text with $\epsilon_{p2b} = 1$



(b) Results for logos with text with $\epsilon_{p2b} = 2$

Figure 5.34: Results of logos with text for $\epsilon_{rdp} \in \{0.1, 0.2, 0.3, \cdots, 1.0\}$ and $\epsilon_{p2b} \in \{1, 2\}$

(a) Results for logos with text with $\epsilon_{p2b} = 3$



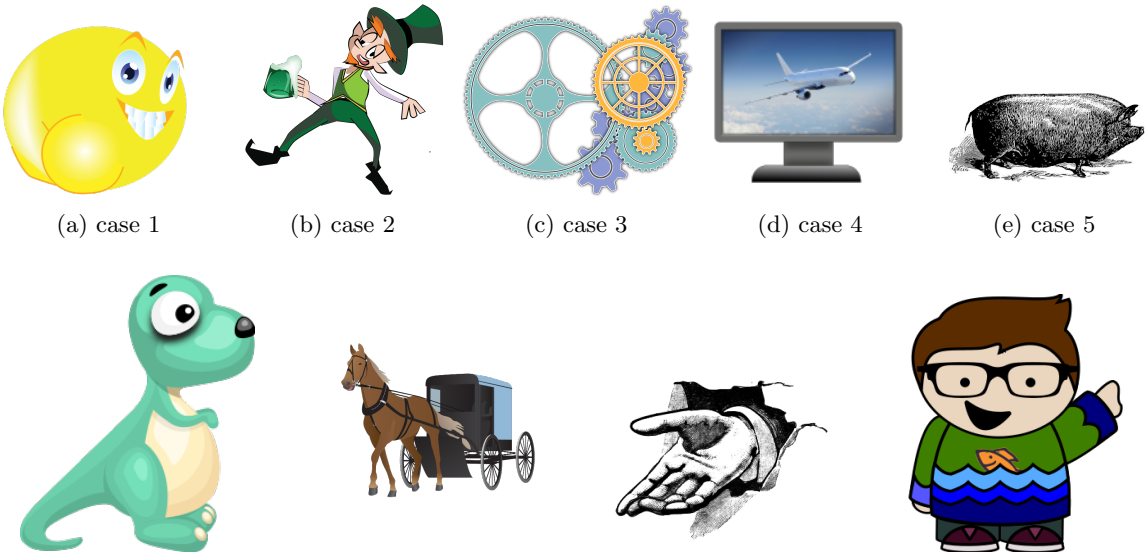
(b) Results for logos with text with $\epsilon_{p2b} = 4$

Figure 5.35: Results of logos with text for $\epsilon_{rdp} \in \{0.1, 0.2, 0.3, \cdots, 1.0\}$ and $\epsilon_{p2b} \in \{3, 4\}$

5.5 Images with embedded image

All cases in this category can be seen in Figure 5.36. Figures 5.43 and 5.44 show the results for images with an embedded image. The permutations of $(\epsilon_{rdp}, \epsilon_{p2b})$ per category are listed in Figure 5.42. For this category, we evaluate the following results:

- **'Excellent' results:** case 8
- **'Good' results:** case 6, as file size is heavily influenced by changing ϵ_{p2b} values; case 9, as there is a sudden drop in quality for permutations with $\epsilon_{p2b} > 2$ and $\epsilon_{rdp} > 0.4$
- **'Bad' results:** case 4, as it performs the worst; case 3, as it is the only case in this category to produce no adequate result at all



(a) case 1 (b) case 2 (c) case 3 (d) case 4 (e) case 5

(f) case 6 (g) case 7 (h) case 8 (i) case 9

Figure 5.36: Dataset images in the category 'images with embedded image'

5.5.1 'Excellent' results

Figure 5.37 shows the result of the only 'excellent' case in this category. It is hard to spot any difference between the images. In terms of compression, there is little difference. After compression, the image is a meager 0.55% smaller. By removing the embedded image element from the SVG image, we observe that there is very little actual path data in the file to compress.

(a) Original image (b) Compressed image (c) Diff image (d) Embedded image removed

Figure 5.37: Result for case 8 with $\epsilon_{rdp} = 1.0, \epsilon_{p2b} = 2, \delta_{fs} = 0.55\%$

5.5.2 'Good' results

The results of case 6 are shown in Figure 5.38. Upon close inspection, subtle changes in the color gradient lines can be observed. Higher values for ϵ_{p2b} allow these gradient lines to be less accurate in resembling the original lines and may account for the increased file size difference.

(a) Original image (b) $\epsilon_{p2b} = 2, \delta_{fs} = 3.53\%$ (c) $\epsilon_{p2b} = 3, \delta_{fs} = 7.22\%$ (d) $\epsilon_{p2b} = 2, \delta_{fs} = 8.61\%$

Figure 5.38: Results for case 6 with $\epsilon_{rdp} = 0.1$

Figure 5.39 shows the results for case 9 with $\epsilon_{rdp} = \{0.4, 0.5\}$. The big drop in SSIM index value is caused by the waves in the shirt. Apparently, the curve fitting does not work anymore with this permutation. This is likely due to too many points being discarded during the Ramer-Douglas-Peucker algorithm, which disables a curve from being fitted properly. With a tighter error margin for the curve fitting this is not a problem, since the curves get broken up in smaller segments.

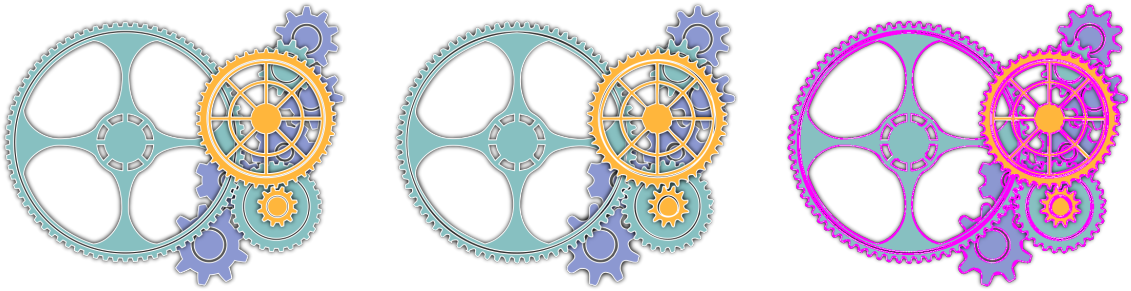
(a) Original image (b) $\epsilon_{rdp} = 0.4$ (c) $\epsilon_{rdp} = 0.5$

Figure 5.39: Results for case 9 with $\epsilon_{p2b} = 3$

5.5.3 'Bad' results

Case 3 is shown in Figure 5.40. Although the edges of the gears are indeed visibly distorted, the overall shape of the image is still fairly decent. If the algorithm can be adjusted to detect objects like the cogs of the gear, images like this case would yield far better results.

The worst result of this category is shown in Figure 5.41. The prototype fails to recognise the sharp corners that do not need to be compressed. Why the big bumps are produced needs to be investigated.



(a) Original image (b) Compressed image (c) Diff image

Figure 5.40: Result for case 3 with $\epsilon_{rdp} = 1.0, \epsilon_{p2b} = 4$

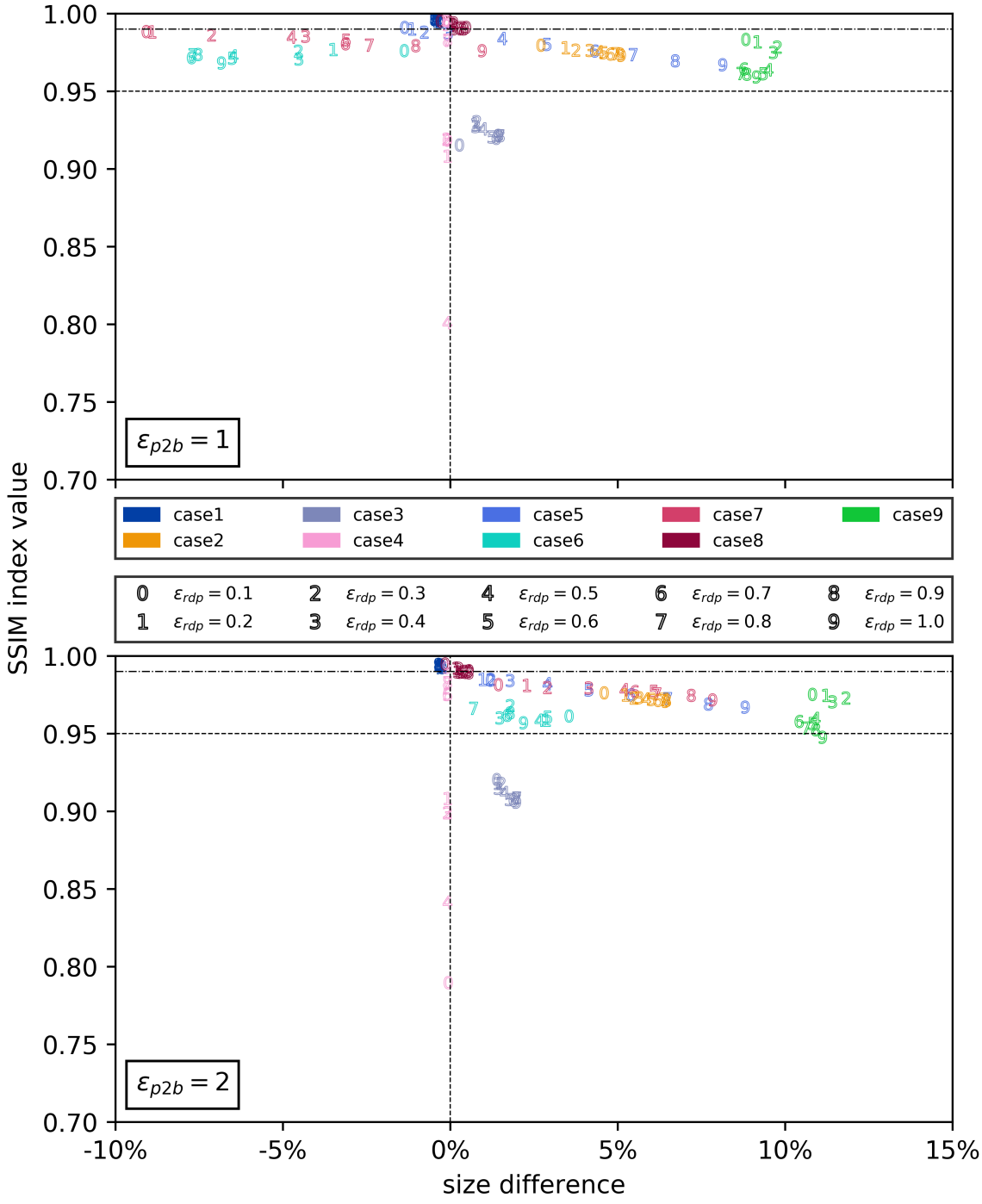
(a) Original image (b) Compressed image (c) Diff image

Figure 5.41: Result for case 4 with $\epsilon_{rdp} = 1.0, \epsilon_{p2b} = 4$

	'Excellent' ϵ_{p2b} values			**'Good' ϵ_{p2b} values**			
	1	**2**	**{3,4}**	**1**	**2**	**3**	**4**
case2				$\{.1, ..., 1.\}$	$\{.1, ..., 1.\}$	$\{.1, ..., 1.\}$	$\{.1, ..., 1.\}$
case5				$\{.5, ..., 1.\}$	$\{.1, ..., 1.\}$	$\{.1, ..., 1.\}$	$\{.1, ..., 1.\}$
case6					$\{.1, ..., 1.\}$	$\{.1, .5, ..., .8, .9\}$	$\{.1, .3\}$
case7				$\{1.\}$	$\{.1, ..., 1.\}$	$\{.1, ..., 1.\}$	$\{.1, ..., 1.\}$
case8	$\{.3, ..., 1.\}$	$\{.2,.3,.8,1.\}$	$\{.2, .3\}$		$\{.4, .7, ..., .9\}$	$\{.4, ..., 1.\}$	$\{.4, ..., 1.\}$
case9				$\{.1, ..., 1.\}$	$\{.1, ..., .9\}$	$\{.1, ..., .4\}$	$\{.1, ..., .4\}$

Figure 5.42: Permutations of $(\epsilon_{rdp}, \epsilon_{p2b})$ for images with embedded image in their respective categories. The cells contain the values for ϵ_{rdp}

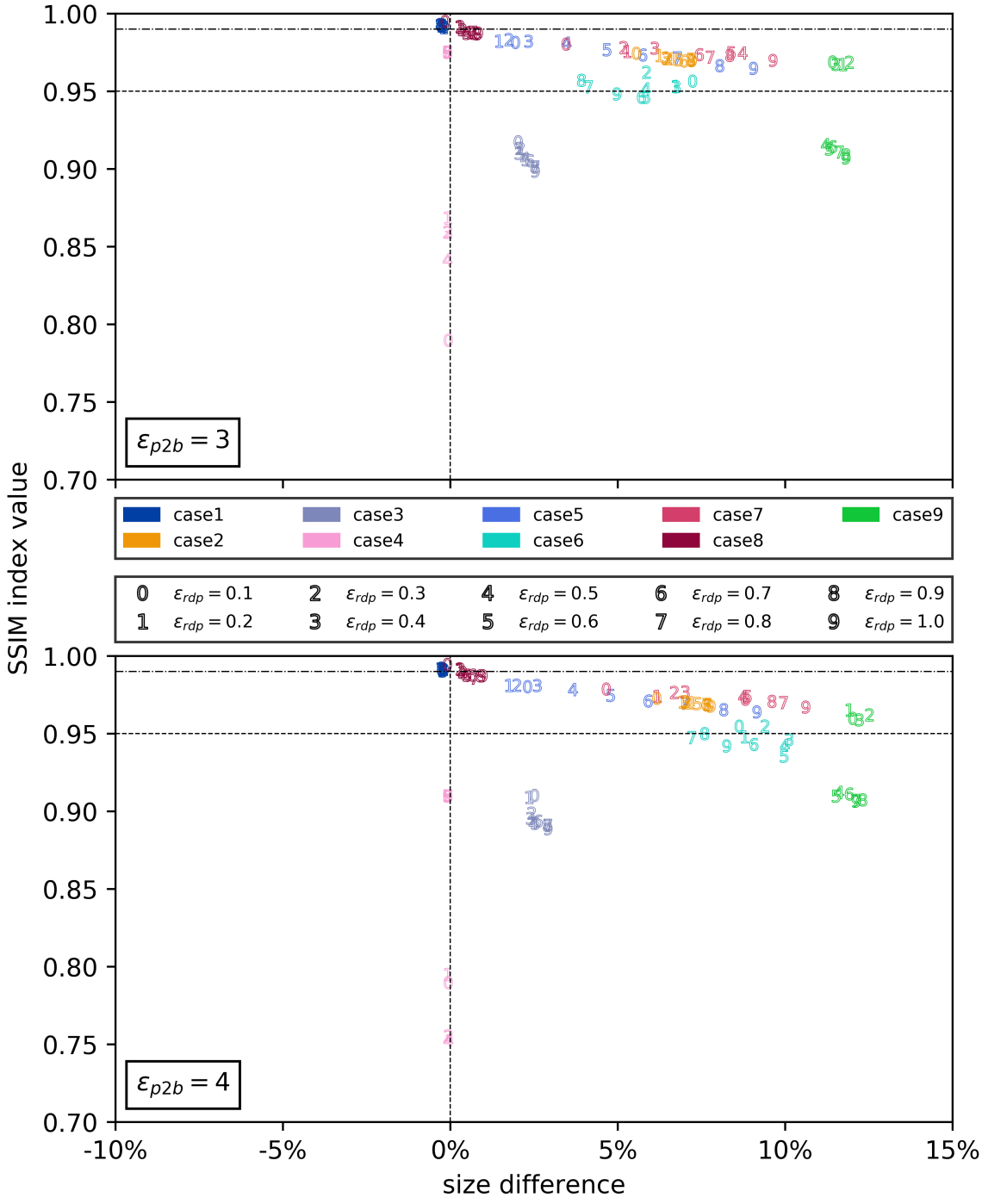
(a) Results for images with embedded image with $\epsilon_{p2b} = 1$



(b) Results for images with embedded image with $\epsilon_{p2b} = 2$

Figure 5.43: Results of images with embedded image for $\epsilon_{rdp} \in \{0.1, 0.2, 0.3, \cdots, 1.0\}$ and $\epsilon_{p2b} \in \{1, 2\}$

(a) Results for images with embedded image with $\epsilon_{p2b} = 3$



(b) Results for images with embedded image with $\epsilon_{p2b} = 4$

Figure 5.44: Results of images with embedded image for $\epsilon_{rdp} \in \{0.1, 0.2, 0.3, \cdots, 1.0\}$ and $\epsilon_{p2b} \in \{3, 4\}$

Chapter 6

Discussion

This chapter discusses the results of the research. The results of the experiments are discussed in Section 6.1. Future work is explored in Section 6.2.

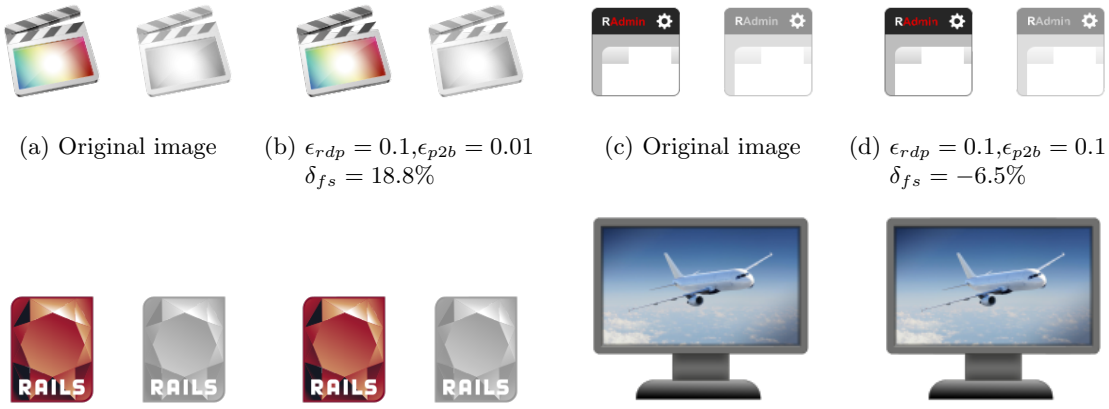
6.1 Prototype performance

The presented results show that the proposed solution is able to produce adequate results. The successful results align with the expectation that images with many curves, such as clipart images, are prime candidates for compression. However, the solution currently suffers from severe limitations that prevents successful compression of all image categories. The main culprit in failing cases seems to be a high grade of straight lines in a path.

When the prototype parses an SVG file, the number of cubic Bézier curves in each path is counted. This number is divided by the total number of elements in the path. If the result is lower than 0.2, the prototype marks it ineligible for compression, as there is too little curve data to gain significant results. However, unsuccessful results like Figure 5.10 indicate that this threshold might be too strict. A sample run with the threshold increased did indeed result in a correct image. The output file size, however, was the same as the input file size. Regardless of the fact that no compression takes place, this does bring the prototype one step closer to a 'one-size-fits-all' solution. If no compression takes place, the SSIM index value is always 1.0. By definition, lossy compression cannot produce an output file identical to the input file while also reducing file size. Hence, if the SSIM index value indicates an identical image, the prototype can report that compression is not feasible for the input image.

With respect to cases as shown in Figures 5.21, 5.31, 5.32 and 5.41, we observed similarities between the problematic areas in these images. These areas mainly have rectangular shapes with rounded corners. These corners cause the straight lines to suddenly display bumps and dents. The root cause of this outcome is found to be in the error margin for curve fitting. Some sample runs with values for $\epsilon_{p2b} < 1$ displayed improvements in fidelity. Figure 6.1 shows a few examples of these improvements.

If a path contains no curves whatsoever, like in Figures 5.20 and 5.30, no loss of fidelity occurs, but the file size also increases. This behaviour is unexpected, as the algorithm copies SVG elements without alterations when those elements are not eligible for compression. Upon closer inspection, we found that the non-gzipped files were actually identical in size and indeed contained identical data. The only difference between the input and output files were the order of tag attributes, such as `height` and `width` for the `svg` tag. The order of these tags is arbitrary and always produces the same result, regardless of what attribute comes first. The library that we use for parsing SVG, pysvg, is also used to write the output file to disk. The pysvg library stores tag attributes in a dictionary assigned to each parsed tag element. Python dictionaries store key-value pairs in arbitrary ways; that is, there is no way to know in what order the keys are iterated over. Hence, when the output file is written to disk, iterating over the attribute dictionary does not necessarily reproduce the attributes in the same order. While this does not pose any difference for a non-gzipped file, the attribute order may impact a file compressed with gzip as observed in this case. The explanation for this phenomenon is that gzip uses a fixed window for referencing to previous occurrences [2]. This is necessary to prevent references



(a) Original image (b) $\epsilon_{rdp} = 0.1$,$\epsilon_{p2b} = 0.01$ $\delta_{fs} = 18.8\%$ (c) Original image (d) $\epsilon_{rdp} = 0.1$,$\epsilon_{p2b} = 0.1$ $\delta_{fs} = -6.5\%$

(e) Original image (f) $\epsilon_{rdp} = 0.1$,$\epsilon_{p2b} = 0.1$ $\delta_{fs} = 4.85\%$ (g) Original image (h) $\epsilon_{rdp} = 0.1$,$\epsilon_{p2b} = 0.1$ $\delta_{fs} = -0.12\%$

Figure 6.1: Better fidelity results for lower ϵ_{p2b} values

to previous occurrences so distant from the current occurrence that the reference becomes larger than the actual segment it is replacing. In this specific case, fewer segments fall within the same window and thus remain uncompressed by gzip. For larger files, this difference becomes negligible. However, for smaller files like these logos, it can amount to several percentages.

It is difficult to determine a single optimal permutation of error margins that can be generalised to compress images with. Some images can endure very lenient error margins and still produce acceptible results. Especially images with many short, non-straight lines can undergo quite drastic changes in fidelity while still maintaining a similar overall appearance. On the other hand, images such as logos tend to require strict error margins to prevent 'iconic' details of the logo being lost in the compression process.

Finally, the results of SVG images with an embedded image show some images simply do not contain enough compressable data to make it worthwile. The SVG parser could be extended to keep track of the length of `image` tags. If the total length of all image tags exceeds a certain threshold, the compression process can be aborted on the premise that the possible size difference will be too marginal to make it worthwhile.

6.2 Future work

Based on the experiment results, the proposed solution can be improved on determining what parts of the input image is eligible for compression. The fidelity of the output file is paramount, and ultimately, the algorithm should always produce a result that is acceptable in terms of fidelity. Whether or not the file size reduction is significant, or even present at all, should not take precedence over fidelity. It has become clear that our proposed method of lossy compression should not be regarded as traditional raster image compression. In that field, the overall fidelity of the image is compromised, and thus the goal is to find the right balance between fidelity loss and file size reduction. In SVG compression, the fidelity of specific elements of an image are compromised while others remain unchanged. Better results can be achieved once key elements are more accurately identified.

The current approach to identifying those elements is quite coarse. One of the consequences of this coarse approach is that the algorithm is not able to discern between text and non-text elements. To the best of our knowledge, a dedicated method to do so does not yet exist. A possible approach would be to apply OCR to individual path elements and mark them as ineligible for compression if text is recognised, although this would be a slow and error-prone approach. A less user-friendly but more rigorous approach would be to add a custom attribute to paths containing text. For example, a simple `contains-text="true"` can be added to those paths. The parser can then mark the path as ineligible.

The current method of comparing input and output files is decent at best. The fact that the files have to be converted to PNG before comparing them makes it unavoidable that some detail is lost. Furthermore, we found that one case was not properly converted by Cairo. The SSIM index value proved to be sufficient to evaluate the results of the experiment, but it is by no means perfect for the job. Future work should implement a better method of comparing SVG files. Further development of comparison based on semantic features could yield such a method.

Chapter 7

Conclusion

In this research, we have introduced a method of compressing SVG images. The method uses existing algorithms and tools in a pipeline setup to achieve compression. To the best of our knowledge, no comparable method has been proposed so far with the goal of compressing SVG images.

The method proved to be effective for some test cases. For some cases the benefits were negligible, and a group of test cases was impacted negatively. We have identified the cause of poor results for specific test cases. Future research should focus on addressing these causes.

To conclude this research, we will answer the research questions from Section 1.5. First, we address research question 4:

RQ 4. *How can existing knowledge on image compression be applied to SVG?*

We found that existing knowledge on image compression is not very relevant to the domain of SVG compression. This is due to the fundamental difference in how these image types work. The method of validating the experiment results, however, is based on Voormedia's regression tests. In those tests, the same method is used to validate the results of their raster-based compression tools.

Next, we answer research question 2:

RQ 2. *When is a reduction in fidelity worthwhile?*

The definition of a 'worthwile reduction in fidelity' was taken from a research that also incorporated the SSIM index value. In this research, a mapping was made between scores given by test subjects and the SSIM index value range that corresponds with these ratings. We validated our results with the two best categories, 'excellent' and 'good'.

With this definition, we can answer research question 1:

RQ 1. *Can lossy compression be applied to SVG images, making the reduction in fidelity worthwhile?*

We found that lossy compression can indeed be applied to SVG images without significantly compromising image fidelity. However, the input image must conform to certain traits to produce successful compression results. With our proposed solution, it is not reasonable to expect positive results for any input image.

The last research question remaining is research question 3:

RQ 3. *Can the algorithm parameters be approximated for optimal compression results?*

This question remains unanswered. The results of the experiment in this research did not show recurring patterns that could be used for parameter approximation. At best, we can suggest to use stricter parameters for compression, as this produces a more similar output file.

Bibliography

[1] Mohamed S Abdelfattah, Andrei Hagiescu, and Deshanand Singh. Gzip on a chip: High performance lossless data compression on FPGAs using OpenCL. In *Proceedings of the International Workshop on OpenCL 2013 & 2014*, page 4. ACM, 2014.

[2] Mark Adler. algorithm - does the order of data in a text file affects its compression ratio? `https://stackoverflow.com/questions/14881312/does-the-order-of-data-in-a-text-file-affects-its-compression-ratio/14885821#14885821`. (Accessed on 07/27/2017).

[3] Wolfgang Boehm, Gerald Farin, and Jürgen Kahmann. A survey of curve and surface methods in CAGD. *Computer Aided Geometric Design*, 1(1):1–60, 1984.

[4] Wolfgang Boehm and Andreas Müller. On De Casteljau's algorithm. *Computer Aided Geometric Design*, 16(7):587–605, 1999.

[5] Eduard Braun. Scour - an SVG scrubber. `https://github.com/scour-project/scour`. (Accessed on 06/24/2017).

[6] Ken Cabeen and Peter Gent. Image compression and the discrete cosine transform. `http://www.lokminglui.com/dct.pdf`. Accessed on (6/20/2017).

[7] Alesandro Cecconi and Martin Galanda. Adaptive zooming in web cartography. In *Computer Graphics Forum*, volume 21, pages 787–799. Wiley Online Library, 2002.

[8] NumPy community. NumPy. `http://www.numpy.org/`. (Accessed on 07/19/2017).

[9] Erik Dahlström, Jon Ferraiolo, Jun Fujisawa, Doug Schepers, Patrick Dengler, Dean Jackson, Anthony Grasso, Cameron McCormack, Jonathan Watt, and Chris Lilley. Scalable vector graphics (SVG) 1.1 (second edition). W3C recommendation, W3C, August 2011. http://www.w3.org/TR/2011/REC-SVG11-20110816/.

[10] Erik Dahlström, Doug Schepers, Cameron McCormack, Jonathan Watt, Chris Lilley, Nikos Andronikos, Rossen Atanassov, Tavmjong Bah, Amelia Bellamy-Royds, Brian Birtles, Bogdan Brinza, Cyril Concolato, Dirk Schulze, Richard Schwerdtfeger, and Satoru Takagi. Scalable vector graphics (SVG) 2.0. W3C recommendation, W3C, sept 2016. http://www.w3.org/TR/SVG2/.

[11] Eugenio Di Sciascio, Francesco M Donini, and Marina Mongiello. A logic for SVG documents query and retrieval. *Multimedia Tools and Applications*, 24(2):125–153, 2004.

[12] David H Douglas and Thomas K Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10:112–122, 1973.

[13] David Duce, Ivan Herman, and Bob Hopgood. Web 2D graphics file formats. In *Computer Graphics forum*, volume 21, pages 43–64. Wiley Online Library, 2002.

[14] Mordy Golding. *Real World Adobe Illustrator CS4*. Peachpit Press, 2008.

[15] Paul Heckbert and Michael Garland. Multiresolution modeling for fast rendering. In *Graphics Interface*. Canadian Information Processing Society, 1994.

[16] Paul S Heckbert and Michael Garland. Survey of polygonal surface simplification algorithms. Technical report, School of Computer Science, Carnegie Mellon University, 1997.

[17] Alain Hore and Djemel Ziou. Image quality metrics: PSNR vs. SSIM. In *Pattern recognition (icpr), 2010 20th international conference on*, pages 2366–2369. IEEE, 2010.

[18] Kaiyuan Jiang, Zhiyuan Fang, Yuanting Ge, and Yu Zhou. Information retrieval through SVG-based vector images using an original method. In *e-Business Engineering, 2007. ICEBE 2007. IEEE International Conference on*, pages 183–188. IEEE, 2007.

[19] Mark Lutz. *Programming Python: Powerful Object-Oriented Programming*. "O'Reilly Media, Inc.", 2010.

[20] Kerim Mansour. pysvg 0.2.2 : Python package index. `https://pypi.python.org/pypi/pysvg/0.2.2`. (Accessed on 07/19/2017).

[21] David Mulder and Curtis Welborn. Lessons in converting from Python to C++. *Journal of Computing Sciences in Colleges*, 29(2):49–57, 2013.

[22] Andreas Neumann and Andréas M Winter. Time for SVG: towards high quality interactive web-maps. *International Cartographic Association*, 2001.

[23] Extensible graphics with SVG. `http://archive.oreilly.com/pub/a/network/2000/04/28/feature/svg.html`. (Accessed on 06/21/2017).

[24] Ananthanarayanan Parasuraman, Valarie A Zeithaml, and Arvind Malhotra. ES-QUAL: a multiple-item scale for assessing electronic service quality. *Journal of Service Research*, 7(3):213–233, 2005.

[25] Zhong-Ren Peng and Chuanrong Zhang. The roles of geography markup language (GML), scalable vector graphics (SVG), and web feature service (WFS) specifications in the development of internet geographic information systems (GIS). *Journal of Geographical Systems*, 6(2):95–116, 2004.

[26] Volker Poplawski. fitCurves: Python implementation of Philip J. Schneider's "algorithm for automatically fitting digitized curves" from the book "graphics gems". `https://github.com/volkerp/fitCurves`. (Accessed on 06/26/2017).

[27] Andy Port. svgpathtools 1.3.1 : Python package index. `https://pypi.python.org/pypi/svgpathtools`. (Accessed on 07/19/2017).

[28] Antoine Quint. Scalable vector graphics. *IEEE MultiMedia*, 10(3):99–102, 2003.

[29] Aleksas Riškus. Approximation of a cubic Bézier curve by circular arcs and vice versa. *Information Technology and Control*, 35(4), 2006.

[30] Alan Saalfeld. Topologically consistent line simplification with the Douglas-Peucker algorithm. *Cartography and Geographic Information Science*, 26(1):7–18, 1999.

[31] Philip J. Schneider. GraphicsGems/FitCurves.c. `https://github.com/erich666/GraphicsGems/blob/master/gems/FitCurves.c`. (Accessed on 07/19/2017).

[32] Philip J. Schneider. An algorithm for automatically fitting digitized curves. In Andrew S. Glassner, editor, *Graphics Gems*, pages 612–626. Academic Press, 1990.

[33] Thomas W Sederberg. Computer aided geometric design. *Computer Aided Geometric Design Course Notes*, 2012.

[34] Wenzhong Shi and ChuiKwan Cheung. Performance evaluation of line simplification algorithms for vector generalization. *The Cartographic Journal*, 43(1):27–44, 2006.

[35] Lev Solntsev. SVGO: Nodejs-based tool for optimizing SVG vector graphics files. `https://github.com/svg/svgo`. (Accessed on 06/24/2017).

[36] Stefan Van der Walt, Johannes L Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D Warner, Neil Yager, Emmanuelle Gouillart, and Tony Yu. scikit-image: image processing in Python. *PeerJ*, 2:e453, 2014.

[37] Guido Van Rossum and Fred L Drake. *The Python language reference manual*. Network Theory Ltd., 2011.

[38] Basic shapes, SVG 1.1 (Second Edition). `https://www.w3.org/TR/SVG11/shapes.html`. (Accessed on 06/21/2017).

[39] Clipping, masking and compositing, SVG 1.1 (Second Edition). `https://www.w3.org/TR/SVG11/masking.html`. (Accessed on 06/21/2017).

[40] Document structure, SVG 1.1 (Second Edition). `https://www.w3.org/TR/SVG11/struct.html`. (Accessed on 06/21/2017).

[41] Filter effects, SVG 1.1 (Second Edition). `https://www.w3.org/TR/SVG11/filters.html`. (Accessed on 06/21/2017).

[42] Gradients and patterns, SVG 1.1 (Second Edition). `https://www.w3.org/TR/SVG11/pservers.html`. (Accessed on 06/21/2017).

[43] Implementation requirements, SVG 1.1 (Second Edition). `https://www.w3.org/TR/SVG11/implnote.html#ArcImplementationNotes`. (Accessed on 06/25/2017).

[44] Minimizing SVG file sizes SVG 1.1 (Second Edition). `https://www.w3.org/TR/SVG/minimize.html`. (Accessed on 07/18/2017).

[45] Paths, SVG 1.1 (Second Edition). `https://www.w3.org/TR/SVG11/paths.html`. (Accessed on 06/21/2017).

[46] Text, SVG 1.1 (Second Edition). `https://www.w3.org/TR/SVG11/text.html`. (Accessed on 06/21/2017).

[47] Zhou Wang. The SSIM index for image quality assessment. `https://ece.uwaterloo.ca/~z70wang/research/ssim/`. (Accessed on 08/15/2017).

[48] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[49] Carl Worth and Behdad Esfahbod. Cairo. `https://www.cairographics.org/`. (Accessed on 07/20/2017).

[50] Thomas Zinner, Osama Abboud, Oliver Hohlfeld, Tobias Hossfeld, and Phuoc Tran-Gia. Towards QoE management for scalable video streaming. In *21th ITC Specialist Seminar on Multimedia Applications-Traffic, Performance and QoE*, pages 64–69, 2010.