Handling Hand-Drawn Diagrams

Ruiting Cao

Master of Science
School of Informatics
University of Edinburgh
2017

Abstract

ZX-calculus is a powerful diagrammatic language to reason about quantum theory. To make drawing the diagrams easier, this project aims to develop a recognition program for hand-drawn ZX-calculus diagrams which could be implemented in a web page or theory rewrite program such as Quantomatic.

The recognition algorithm is a hypothesis-based system divided into four stages including segmentation, classification, connection and correction. The strokes drawn by the users are represented by paths in the SVG format. We segment the hand-drawn diagram into path groups and generate hypotheses representing different ways of combinations. When the users correct the diagram, it will generate more hypotheses candidates. Each path group forms a shape and then the shapes are fed into an SVM classifier for classification. Finally, we connect the recognised elements with the diagram rules and score the hypotheses to find a winner.

We visualise the recognition results in SVG images and LaTeX PDF to evaluate the accuracy. The accuracy of uncorrected diagrams and corrected diagrams are 94.33% and 69.26%. The failures are mainly caused by unconventional user behaviours and the defective rules for determining users' corrections. According to the evaluation results, the future work will primarily focus on improving hypothesis mechanism and applying more professional domain knowledge to the diagram rules.

Acknowledgements

First and foremost, I would like to thank my supervisor Dr Chris Heunen. He gave me so much really useful advise on the algorithm design and and thesis writing. He was always there when I have problems with the project. Further more, Thanks are also due to Xi Huang, my reliable partner of this project who worked with me together. We enjoyed a lot of good time. Finally, I would like to thank my loved ones who supported me throughout the entire process.

Declaration

Table of Contents

Chapter 1

Introduction

1.1 Motivation

The ZX-calculus diagram is a powerful graphical language for reasoning about quantum theory. This formalism is intuitive for expressing the networked process. As quantum computation becomes a more mature technology now rather than pure theories, the demand of using ZX-calculus also increases (de Beaudrap and Horsman, 2017). But it is inefficient to draw an electronic form of the diagrams on a computer with the current human-computer interface. Users need to spend time on choosing and learning the drawing tools first and then generate the diagrams by typing commands or doing the click-drag motion.

There are some recognition algorithms for diagram sketches in the past few years. However, each kind of diagrams has its features, so these methods only work for specific diagrams. This is the reason why we need to develop a new algorithm for recognising ZX-calculus sketches in this program.

1.2 Hypothesis and Deliverables

Our goal is to design an algorithm to recognise hand-drawn ZX-calculus diagrams and do the implementation, so the hypothesis of this project is whether such recognising algorithm exists. The deliverable program should have three core functions which include recognising all the elements in a sketch, connecting these elements to a diagram, and allowing users to correct their drawing.

The final output of the program is some structured data containing the recognised diagram information. The data should be able to help reconstruct the diagram sketch or to be used in other applications.

1.3 Contributions

The main contribution of this project is that we provide an efficient recognition program for ZX-calculus sketches. The output of this program can be embedded in other applications for different uses. For instance, it can be embedded into web page or software, where users can get LaTeX code of the diagram sketch and also PDF images generated by this code. They might use these images either for papers or presentations. Besides, it can also assist quantum graphical reasoning programs such like Quantomatic (Kissinger, 2012), which is an automatic rewrite program that translates ZX-calculus diagrams to other formalisms given rewrite rules. With our recognition algorithm, users no longer can import the diagrams with much less time and efforts.

Although this project focus on ZX-calculus diagrams, we try to design our algorithm to be as general as possible. The recognition program is also extensible for other string diagrams such like data flow diagrams. It allows the wire elements to have a very high flexibility in their shape, as we separate the wire elements and node elements before recognising them. And we use a feature-based classifier which enables other developers to modify or add node types quickly.

Chapter 2

Background and Related Work

2.1 ZX-Calculus

ZX-calculus is a graphical language based on category theory introduced by Coecke and Duncan (Coecke and Duncan, 2011). Comparing with the low-level formalism such as matrix mechanics, the graphical formalism is more intuitive and easier for parsing complex computations.

A ZX-calculus diagram is a kind of string diagram consisting of nodes and wires. Nodes denote maps and wires denote systems. In different ZX-calculus version, their appearance might have small variations. These are some commonly used elements in the diagram:

- Dot nodes. There are two colours of dot nodes, usually green and red. The green nodes represent X-basis elements, and the red ones represent Z-basis elements. They can have an arbitrary number of inputs and outputs including none.

- Morphism nodes. The morphism node appears as a right trapezoid with a function label. It has four orientations. Like the dot nodes, morphism nodes also have multiple inputs and outputs. The input and output wires are connected to the trapezoid's top side and bottom side.

- H nodes. The H nodes are yellow squares with exactly one input and one output. It represents the Hadamard gate used for changing the X and Z basis.

- Black diamond. The black diamonds don't have any inputs or outputs. It is the result of composing a cup and a cap.

- Wires. The wire ends at the input or output interface point of the diagram. Wires linking inputs form a cup and wires connecting outputs form a cap. They can be bending or straight and can intersect with other wires.

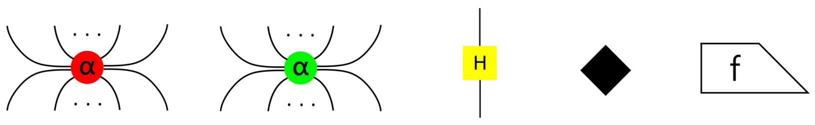


Figure 2.1: ZX-calculus elements.

We read a ZX-calculus diagram from bottom to top. The parallel operations lie horizontally in the diagram, and the sequential operations are stacked vertically. There are some other nodes and diagram rules to denote more complicated computation. However, in this thesis, we only consider the diagrams with dot nodes, morphism nodes and wires with the basic rules mentioned above.

2.2 Image Format

The inputs of a diagram recognition algorithm are diagram images. We considered two image formats and compared their features on recognising sketches.

2.2.1 Pixel-based graphics

We first consider pixel-based graphics such like Portable Network Graphics (PNG). These are raster images composed of pixels. The strokes drawn by users are black dots in pixel slots. With pixel-based graphics, recognition is a process dealing with the matrix of pixel values.

- Advantages: ① Most previous sketch recognition study used pixel-based images as inputs, which could provide many useful recognising ideas and tools. For example, feature extraction techniques were frequently used to find out image patterns and were proved successful. ② There are also some great existing libraries for pixel-based image recognition such like OpenCV.

- Drawbacks: ① Users' strokes are important information for separating the elements in the diagram, but it's impossible to get the strokes in pixel images without front-end recording support. ② Although existing methods performed well on shapes, it might be hard to recognise wires, especially long, bending and intersected wires.

2.2.2 Vector-based graphics

We also consider the vector-based graphics such as Scalable Vector Graphics (SVG).Vector graphics lead through locations by control points with definite positions on the x axis and the y axis to determines the paths. Paths could be assigned with different attributes. With vector-based graphics, recognition is to handle these control points.

- Advantages: ① It is straightforward to know how the users draw strokes one by one since they are stored as separate paths. It helps us to divide the strokes into shapes. ② We can use the sequential positions to calculate stroke directions and intersections easily.
- Drawbacks: It's hard to recognise complicated shapes with sparse control points unless we set the smoothness of the strokes at a low value.

In this project, we choose the vector-based graphics SVG as the inputs of the recognition algorithm. First, the nodes in ZX-calculus diagrams are shapes with simple features which don't need powerful shape recognition methods. Second, unlike recognising a single shape, it is essential to separate the elements correctly and find the relationships between them when dealing with a diagram. So SVG performs better in the aspects that we care about.

2.3 Shape Classifier

In the recognition algorithm, we need a classifier to classify the nodes into dots and morphisms. The inputs of the classifier could be shape attributes, path points or path curvatures. Then the feature dimension is likely to range from 2 to 100, and the dataset could be linear or nonlinear data. We compared some mainstream supervised classification models used in sketch recognition (Ouyang and Davis, 2009b; Li et al., 2013; Awal et al., 2014; Angadi and Lakshman Naika, 2014) and finally choose SVM as the node classifier.

2.3.1 Lazy Learning

Lazy learning such like K-Nearest Neighbors (KNN) is a primary classification algorithm. It calculates generalisation of the training set only when there is a request.

The advantage of lazy learning is that it can solve multiple problems simultaneously because the target function is approximated in a local area (Wettschereck et al., 1997). For example, KNN assigns an object to a class by the vote of its k nearest neighbours.

Its disadvantages include we have to prepare a dataset with a large size and the high quality. Lazy learning needs sufficient data makes sure the new query can always find proper neighbours for comparing (Guru et al., 2010). Besides, it is sensitive to outliers because it doesn't calculate generation in training phase or doesn't have the training phase. Another disadvantage is that the processing time of classifying is long especially when the feature dimension is high.

However, lazy learning such like KNN is usually defeated by other models when doing sketch recognition in both accuracy and processing time. The main reasons are a lack of the delicate sketch dataset in a large size and the difficulty in finding the proper window parameter k.

2.3.2 Support Vector Machines

Given a set of labelled points in the space, Support Vector Machines (SVM) can separate these points with one or more hyperplanes which make the gaps between categories as wide as possible. In general, a larger margin means a lower generalisation error. For the data that can't be separated linearly, it uses the kernel trick to map the original space to a higher dimensional space to do the separation.

Compared with lazy learning, the SVM gives a more accurate and robust result because it provides a good out-of-sample generalisation by only using the most relevant points (Auria and Moro, 2008). Besides, it can avoid local minima and provide a unique solution.

The most problem with the SVM is the choice of the kernel and the kernel's parameters. The kernel models are very sensitive to over-fitted model selection criterion (Cawley and Talbot, 2010). Besides, the SVM doesn't have probability estimates itself. These are calculated using an expensive five-fold cross-validation

So the SVM seems a good model to classify nodes in our algorithm as it could provide a rather good result and flexible choices to use different kernels according to the data distribution.It is also the most used model recently for recognising symbols in various 2D languages such like diagram Hammond et al. (2010); Ouyang and Davis (2009a), circuit Angadi and Lakshman Naika (2014) and mathematical expressions Awal et al. (2014). These papers proved that considering precision and computation, the SVM achieved a better result in symbol classification than other models such as KNN, Image Deformable Model, and Neural Network.

2.3.3 Neural Network

Neural Network is a network composed of many interconnected neurons, which receive input, change their states and produce output depending on the input and activation. These neurons are organised in different layers performing different transformations on their inputs.

The most advantage of the neural network is that it can learn imprecise pattern from the examples. As it has the ability approximating any function regardless of the linearity, it is usually used for complicated data (Dreiseitl and Ohno-Machado, 2002).

However, the disadvantage of the neural network is obvious. It spends much longer time on training the dataset than other classifiers we mentioned above (Ge et al., 2013). Sometimes other simpler models can reach the same accuracy as the neural network does and they cost much less.

In sketch recognition studies, the neural network usually uses pixels as the inputs to extract patterns during training instead of extracting features in advance. The testing results showed that the neural network is a powerful model, but we think it's a kind of overuse for this project. Shapes in the ZX-calculus diagrams are simple and don't have many useful patterns for training.

2.4 Related work

There aren't any existing methods on recognising ZX-calculus diagrams, so we searched for some papers that work on other diagrams. Their algorithms can be used for reference when we work on ZX-calculus.

We introduce the methodology of recognising circuits and chemical diagrams. Among all the diagram recognition papers, circuits are the most similar one to the ZX-calculus, because both of them are string diagrams. So we hope to learn some general ideas of string diagram recognition from the papers working on circuits. As for chemical diagrams, they are quite different from ZX-calculus because they don't have connectors (wires). However, the hypothesis system for segmentation introduced in chemical diagram recognition is a good design, and it applies to general diagrams. So we include this method in this section.

Besides the methodology of recognising the whole diagram, we also introduce some work focused on user correction on the diagrams, as we plan to add correct function in our algorithm.

2.4.1 Circuits

The most recent study on circuit recognition is proposed by (Angadi and Lakshman Naika, 2014). This algorithm doesn't need the support of front-end recording. Instead, it uses complete circuit bitmap images as the input. It divides the recognising process into three stages:

- Segmentation: segment strokes or break up the image into shapes
- Classification: classify shapes
- Connection: connect discrete shapes to a diagram

In the segmentation part, it uses pixel density to remove the straight parts of the wires but keep the corners. The remains are dispersed nodes and components, and the corner of the wires. This is more efficient than the previous method of combining the strokes in (Calhoun et al., 2002).

After getting these separated elements, it extracts features from each of them and feeds the data into an SVM classifier. It adopts the methods in (Gennari et al., 2005), use the geometry and simple domain knowledge to connect these recognised elements with wires.

Although this algorithm works for pixel-based diagram images, it provides a good solution to differentiating nodes and wires. It is possible to apply this segmentation algorithm to other string diagrams even if the input images are in the SVG format, as we can use point density instead of the pixel density. However, the nodes in ZX-calculus are simple shapes with low point density, which means such segmentation might mix up the nodes and wires. If so, we will try to combine strokes using the methods in (Calhoun et al., 2002).

2.4.2 Chemical Diagrams

The latest study on chemical diagram recognition is proposed by (Ouyang and Davis, 2011). In this approach, the basic unit in the diagram is corner points. Each two corner points compose a stroke segment, and the stroke segments form a shape element. It uses the conditional random fields to combine the features in corner points, stroke segments and shapes. There aren't the three stages we mentioned in circuit recognition. Instead, the whole recognition is a probability-based system capturing the relationships between the entities. This approach achieves nearly perfect results. However, it is not suitable for ZX-calculus. The reason is, in chemical diagrams, the stroke segments in one shape have regular patterns. But ZX-calculus elements are usually drawn with one stroke and the segments do not have certain patterns.

We find a better system design in (Ouyang and Davis, 2007). It uses the front-end tablet to record stroke sequences and then generates hypotheses of all different combinations of up to 7 sequential strokes. Shape features are extracted from these stroke groups and put into an SVM classifier. By using prior chemical valence knowledge to check the rationality of adjacent elements, the system can find out the most reasonable segmentation hypothesis, and also to recover the diagram from inconsistencies by adding strokes. Apparently, this is a more general approach than the CRF which can be applied to other kinds of diagrams as well. The hypothesis-based system might also work well for ZX-calculus diagrams.

2.4.3 User Correction

Besides the algorithm of recognising a whole diagram image, in Wu et al. (2014) there was an approach to incorporate users' correction on recognised sketches, which could also be applied to diagram recognition. This research discovers three editing modes of users' intention when they're correcting: local correction, replacement, and enhancement. Based on these three editing modes, different correction algorithms are designed. But this paper only focused on single symbol correction and assumed all strokes are already well grouped. Connectors were not considered in this research.

2.5 Conclusion

We discuss the input formats, classifying tools and some related works in the previous sections. Although there is no existing recognition for ZX-calculus diagrams, there are still some good solutions in previous research for other diagrams. To develop our algorithm, first, we decide to choose SVG as the input format. SVG internally records stroke sequences and stores them separately, which could help to do the segmentation. As the ZX-calculus diagram is a kind of string diagram comprised of nodes and wire, we could use the similar three stages used in circuit recognition including segmentation, classification and connection, and build up a hypothesis-based system for segmentation and connection parts.

Chapter 3

Methodology and Implementation

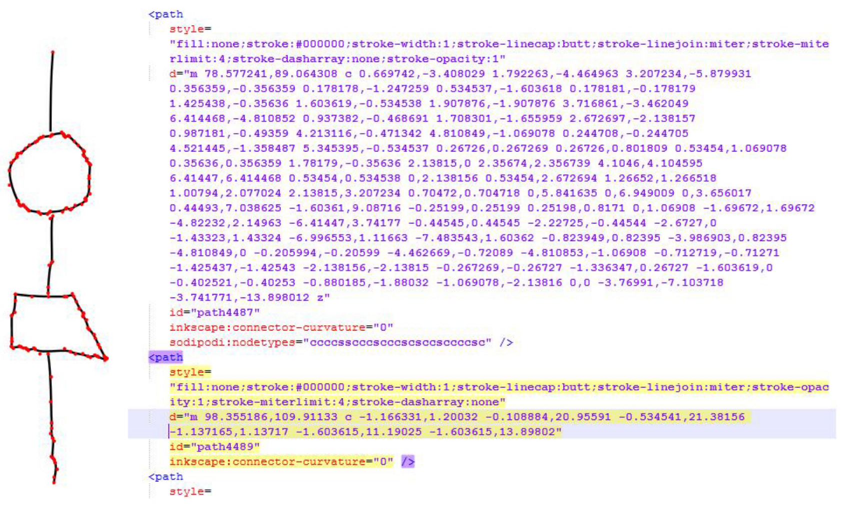
In this chapter, we introduce the methodology of recognising freehand ZX-calculus diagrams. We'll describe the four main stages of the recognition algorithm are segmentation, classification, connection, and correction in detail, and also briefly explain how we realise this algorithm in code.

3.1 Input and Preprocessing

The input of the algorithm is an SVG file containing one ZX-calculus diagram. All the diagram strokes are drawn by freehand lines without snapping in Inkscape and stored as paths separately in the file. As SVG files could be read in Extensible Markup Language (XML), we load the XML tree of the input SVG file and extract all path elements.

The path elements have an attribute "d" which contains point positions. The "$m\ (x,y)$" command means move to the absolute position and the "$c\ (x_1,y_1)\ (x_2,y_2)$..." command means the relative positions of a sequence of Cubic Bezier Curves. A Cubic Bezier Curve has one start point, two control points, and an end point. So each three points in the c command is a group, and the last point of the former group is the start point of the current group. Then we can calculate all the absolute positions in the c command.

In our algorithm, we set the smoothness of the freehand line at a small value so that the strokes can describe the corners of the trapezoids. With such setting, almost all the control points also lie on the shape corner, so we treat them as the start points and end points. Finally, we get set of paths, and each path contains a sequence of point positions.



```
<path
   style=
   "fill:none;stroke:#000000;stroke-width:1;stroke-linecap:butt;stroke-linejoin:miter;stroke-mite
   rlimit:4;stroke-dasharray:none;stroke-opacity:1"
   d="m 78.577241,89.064308 c 0.669742,-3.408029 1.792263,-4.464963 3.207234,-5.879931
   0.356359,-0.356359 0.178178,-1.247259 0.534537,-1.603618 0.178181,-0.178179
   1.425438,-0.35636 1.603619,-0.534538 1.907876,-1.907876 3.716861,-3.462049
   6.414468,-4.810852 0.937382,-0.468691 1.708301,-1.655959 2.672697,-2.138157
   0.987181,-0.49359 4.213116,-0.471342 4.810849,-1.069078 0.244708,-0.244705
   4.521445,-1.358487 5.345395,-0.534537 0.26726,0.267269 0.26726,0.801809 0.53454,1.069078
   0.35636,0.356359 1.78179,-0.35636 2.13815,0 2.35674,2.356739 4.1046,4.104595
   6.41447,6.414468 0.53454,0.534538 0,2.138156 0.53454,2.672694 1.26652,1.266518
   1.00794,2.077024 2.13815,3.207234 0.70472,0.704718 0,5.841635 0,6.949009 0,3.656017
   0.44493,7.038625 -1.60361,9.08716 -0.25199,0.25199 0.25198,0.8171 0,1.06908 -1.69672,1.69672
   -4.82232,2.14963 -6.41447,3.74177 -0.44545,0.44545 -2.22725,-0.44544 -2.6727,0
   -1.43323,1.43324 -6.996553,1.11663 -7.483543,1.60362 -0.823949,0.82395 -3.986903,0.82395
   -4.810849,0 -0.205994,-0.20599 -4.462669,-0.72089 -4.810853,-1.06908 -0.712719,-0.71271
   -1.425437,-1.42543 -2.138156,-2.13815 -0.267269,-0.26727 -1.336347,0.26727 -1.603619,0
   -0.402521,-0.40253 -0.880185,-1.88032 -1.069078,-2.13816 0,0 -3.76991,-7.103718
   -3.741771,-13.898012 z"
   id="path4487"
   inkscape:connector-curvature="0"
   sodipodi:nodetypes="ccccssccсscccscsccsccccsc" />
<path
   style=
   "fill:none;stroke:#000000;stroke-width:1;stroke-linecap:butt;stroke-linejoin:miter;stroke-opac
   ity:1;stroke-miterlimit:4;stroke-dasharray:none"
   d="m 98.355186,109.91133 c -1.166331,1.20032 -0.108884,20.95591 -0.534541,21.38156
   -1.137165,1.13717 -1.603615,11.19025 -1.603615,13.89802"
   id="path4489"
   inkscape:connector-curvature="0" />
<path
   style=
```

Figure 3.1: The left is an example of freehand diagram in SVG. We plot all the points of the paths on the image. The right image is its corresponding XML tree.

3.2 Segmentation

Although we already have every stroke stored in each path separately, it is not reasonable to skip the segmentation phase and classify the strokes directly. There is one thing we must consider is that the user might draw one element by multiple strokes. For example, it is very likely to draw a trapezoid in three strokes. So the first step of our algorithm is still to segment all the strokes into shapes. Our segmentation algorithm could find groups consisting of 2 or 3 strokes with all sorts of matching conditions.

3.2.1 Find Neighbouring Strokes

If the user draws a shape in multiple strokes, the ends of these strokes must be close. We use this trick to find adjacent stroke pairs.

We also record the matching types of every adjacent pair. Each stroke has two ends with different positions, so there are six types of matching, including four types of two-end matching and two types of four-end matching. We create a match list to store these matchings for further use. Then we search matches with three strokes in the match list. For example, there are three path objects named path 1, path2 and path3. If the matches (path1,path2), (path2,path3), (path1,path3) are simultaneously found in the match list, we will create a new match [path1,path2,path3] and append it to the list.

We only implement the algorithm finding up to 3 matching strokes in this pro-

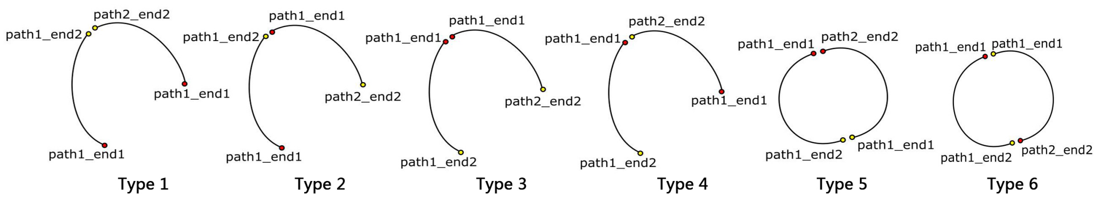


Figure 3.2: The six matching types of two strokes.

gram. Similar methods could be used for searching groups including more strokes.

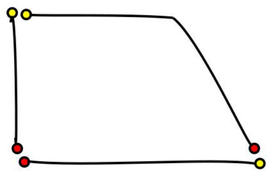


Figure 3.3: The match of three strokes.

Then we could combine these pairing matches to one path. As each path comprises a sequence of coordinate points, we must combine matched strokes according to their matching types. For those groups with more than two strokes, we will combine the strokes two by two.

3.2.2 Generate Hypotheses

After finding out the pairing strokes, we generate all possible segment hypotheses with these matches. Mathematically, there should be 2^n hypotheses with n matches, so we generate a list of n binary number to represent these hypotheses in the program. If

0	1	0	1	1	0	1

Figure 3.4: One of the hypothesis label when there are 7 matches among all the strokes. This hypothesis will switch on the 2nd, 4th, 5th, and 7th match in the match list.

the ith binary number is 1, we will use the ith match in the match list in this hypothesis. Otherwise, we still use the separated strokes. For those matches switched on, the old pairing paths are removed from the diagram and the new combined path is added. One thing needs to be mentioned is that there are some invalid hypotheses.

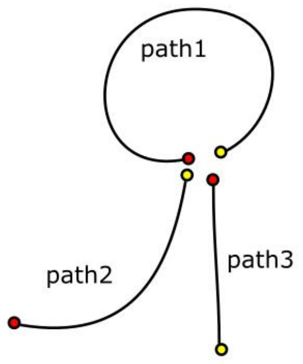


Figure 3.5: This diagram includes four matches: (path1,path2) labeled match1, (path1,path3) labeled match2, (path2,path3) labeled match3 and (path1,path2,path3) labeled match4. There are 2^4 binary numbers represent 16 different hypothesis candidates: "0000", "0001", "0010", "0011", "0100",...,"1111", where only 5 hypotheses are valid: "0000", "0001", "0010", "0100" and "1000".

In Figure 3.5, hypothesis "1100" is invalid because we can't use match1 (path1,path2) and match2 (path1,path3) simultaneously. If (path1,path2) is switched on, (path1,path3) can't be used because path1 is already removed. So we check the validity before using each match. Once one match conflicts with others in the hypothesis, we throw up this hypothesis.

After the segmentation stage, we should get a set of hypotheses where each hypothesis contains grouped paths. Each path is considered a shape and ready for being classified.

3.3 Classification

In this phase, we first transform all the paths to polygons. To describe a shape, we could calculate the shape factor including aspect ratio, compactness, elongation, and waviness. We choose three factors: aspect ratio, compactness, and elongation which can tell circles from trapezoids. We create a dataset of freehand nodes, extract these attributes and use these them as the training data for the SVM classifier.

3.3.1 Classify Wires

For each path in the hypotheses, we don't classify the wires by the SVM classifier, because their shape features have a high randomness. It might confuse the classifier, especially if we add more elements into the diagrams in the future. For this reason, we use two simple but effective rules to exclude the wires from the nodes before we apply the SVM classifier:

- Calculate the distance d between the two ends of the path and the perimeter P of the path. If the the ratio $\varphi_{open} = \frac{d}{P}$. If the φ_{open} is larger than a threshold value, we consider it as an open shape. Otherwise, it is a closed shape. If the path is an open shape, it must be a wire.

- Transform the paths into polygons. Check whether the polygon is a convex shape or a concave shape by calculating its convex ratio $\varphi_{convex} = \frac{area_{shape}}{area_{convexhull}}$. If the φ_{convex} is smaller than a threshold value, it is a concave shape. It is defined as a convex shape otherwise. If the path is a concave shape, it is a wire.

If a path satisfies any one of these two conditions, we classify the path into the wire class. Only closed and convex shapes are classified into nodes.

3.3.2 Classify Nodes

After we exclude the wires, the next step is to classify nodes into dots and morphisms. We adopt a more delicate method: create a dataset and train an SVM classifier, which provides a much higher accuracy than classifying by simple decision rules.

3.3.2.1 Extract Features

We first extract features from the rest polygons transformed from paths. These features include:

- Eccentricity: Find the two principal axes of the shape and get the eigenvalues λ_1 and λ_2.

$$cov = \frac{1}{N}\sum_{i=0}^{N-1}\begin{pmatrix} x_i - g_x \\ y_i - g_y \end{pmatrix}\begin{pmatrix} x_i - g_x \\ y_i - g_y \end{pmatrix}^T = \begin{pmatrix} c_{xx} \; c_{xy} \\ c_{yx} \; c_{yy} \end{pmatrix}$$

cov is the covariance Matrix of the polygon and $G(g_x, g_y)$ is the polygon centroid.

$$\begin{cases} \lambda_1 = \frac{1}{2}[c_{xx} + cyy + \sqrt{(c_{xx}+c_{yy})^2 - 4(c_{xx}c_{yy} - c_{xy}^2)}] \\ \lambda_2 = \frac{1}{2}[c_{xx} + cyy - \sqrt{(c_{xx}+c_{yy})^2 - 4(c_{xx}c_{yy} - c_{xy}^2)}] \end{cases}$$

Then calculate the ratio E of the eigenvalues and use it as a feature.

$$E = \frac{\lambda_1}{\lambda_2}$$

- Circularity: The compactness equation should be:

$$comp = \frac{2\sqrt{a\pi}}{P}$$

However, according to (Montero and Bribiesca, 2009), if the shape has a twisted contour, the compactness value will shrink.

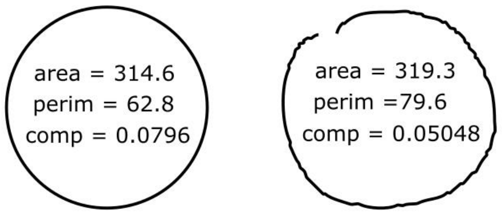


Figure 3.6: The different compactness values of the circles with the same size. It's the twisted contour that causes the shrink of compactness.

As there are usually fewer twists when drawing a trapezoid with straight lines, and more twists when drawing a circle with more curves in an SVG canvas with a low smoothness, it's not smart to use compactness factor directly as a feature of the freehand shapes. So we modify the equation. We calculate the Euclidean distance between the centroid and each vertex and find the farthest vertex away from the centroid point. Use the distance as the radius r. Then calculate the ratio C of the shape area to the circle area with radius r. Then the C is the second feature.

$$C = \frac{a}{\pi r^2}$$

- Aspect Ratio: Find the ratio of the two boundaries. Find the boundary of the shape on x axis (x_1, x_2) and y axis (y_1, y_2). Then calculate the ratio of x boundary and y boundary.

$$A = \frac{x_2 - x_1}{y_2 - y_1}$$

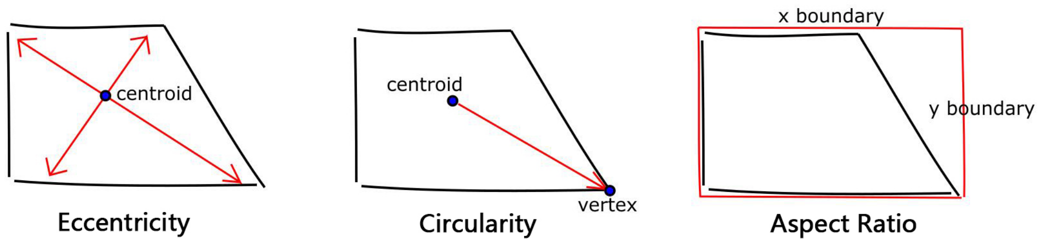


Figure 3.7: This figure shows the three features of a morphism shape.

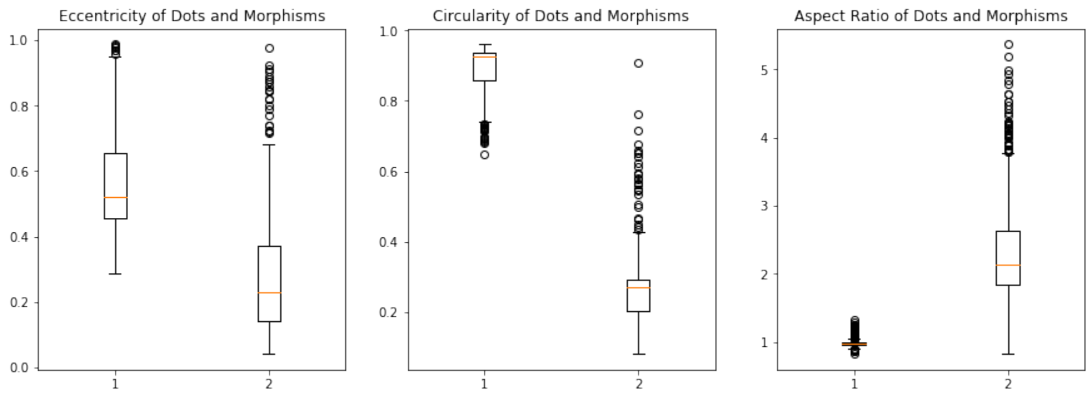


Figure 3.8: This set of box plot to make sure that each of these features have the ability to distinguish the circles and trapezoids.

3.3.2.2 Shape Generation

Before we use the classifier to classify shapes, we need to prepare the training dataset to train it. However, the number of real hand-drawn shapes is insufficient, and it is very likely that sometimes the user's drawing has a bad quality which makes the shapes become outliers. So we decide to write a snippet of code to generate shapes with a stable and good quality. Using code to produce a relatively large size of data is a kind of bootstrapping method, which can help the classifier to find a more accurate distribution of the high-quality shapes.

To automatically generate shapes, we first need to find the equations of a circle and a right trapezoid. According to Figure 3.9, the $(0,0)$ coordinate is on the top-left side of the canvas. For a circle, all the points(x,y) should satisfy $x^2+y^2=r^2$. So we generate a random x value between $-r$ and r and calculate y using the equation. The y will randomly be positive or negative. And then we generate n points and then sort

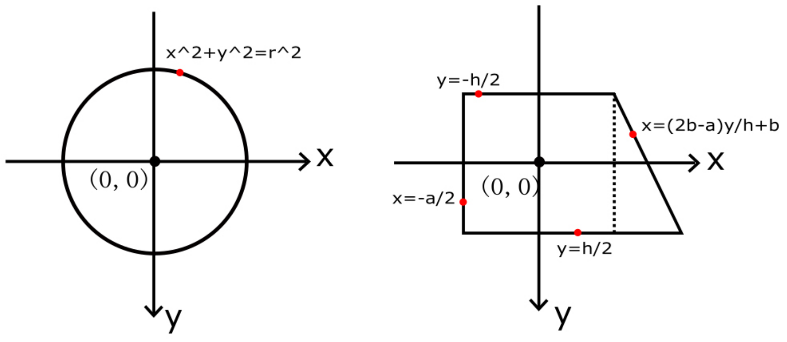


Figure 3.9: The equations of the curves and lines in the shapes.

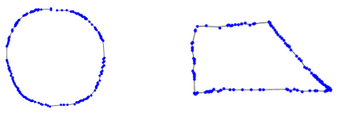


Figure 3.10: Shapes created by real users.

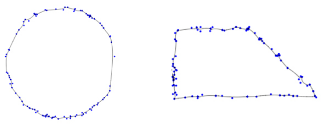


Figure 3.11: Shapes created by code.

them in clockwise. After these steps, we get a sequence of path points.

For a morphism, the points lie on four different lines. Before generating a point, we need to select to which line the point belongs randomly. Then we do the same thing as we did for the circle to create the points of the path. For example, if the point belongs to the line $x = -a/2$, we will set the x value at $-a/2$ and generate a random y value between $-h/2$ and $h/2$.

When we create the circles and right trapezoids, all the parameters including point number n, circle radius r and trapezoid edge lengths a, b, h are arbitrary values in some certain ranges, so that the shapes can have a random size and a random point density.

The final step is to slightly deform the regular shapes to make them look more ”freehand”. For each point on the shapes, we give it a small random movement in both x axis and y axis. Then we get the ”freehand” shapes, and we could use them for the classifier.

3.3.2.3 Train the Classifier

These shape attributes are the training data of the SVM classifier. The feature dimension is three, so we visualise the labeled training data. According to Figure 3.12, there are only a few outliers. The two shapes’ features are distributed nicely that we can directly use a 2d plane to separate these points. So we plan to use a binary SVM with the linear function kernel instead of a nonlinear function kernel.

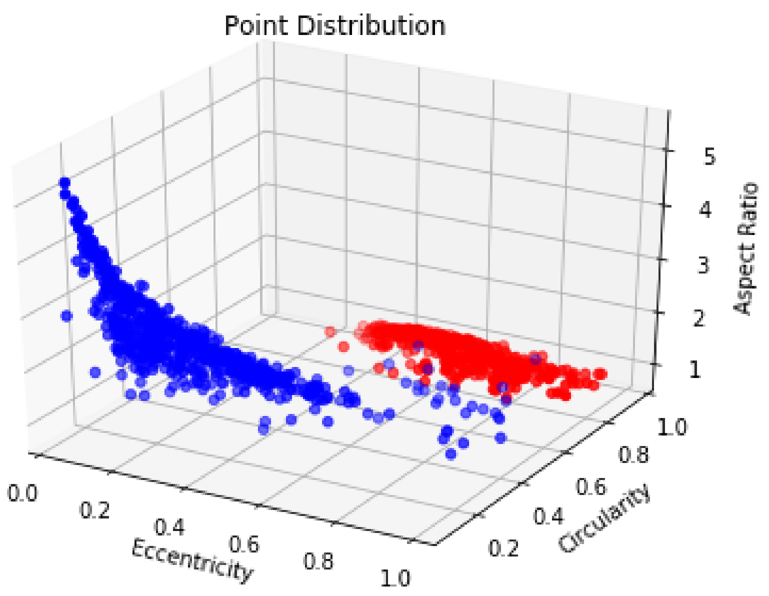


Figure 3.12: The data distribution of 600 dots and 600 morphisms. The three axes are eccentricity, circularity and aspect ratio. The red points represent dot features, and the blue ones represent morphism features.

We set the plane equation as:

$$f(\vec{x}) = \vec{\omega}^T \vec{x} + d$$

where $\vec{x_i} = (x_i, y_i, z_i)$ and $\vec{\omega} = (a, b, c)$. We label dot as $\{+1\}$ and morphisms as $\{-1\}$, and set the red points' side is the positive side to the plane and the blue points' side is the negative side. So the distance from a point to this plane is:

$$dist = \frac{1}{||\vec{\omega}||}(\vec{\omega}^T \vec{x} + d)$$

According to the definition of the SVM, we want to find a 2d plane which makes the largest gap between the two categories. The closest red point and blue point to this plane are called support vector. The distance of two support vectors to the plane is the same. Then the plane should equal to $\arg\min_{plane} margin(plane)$ where function $margin(*)$ means the distance from the support vector to the plane.

So for each point $\vec{x_i}$, we want to find:

$$\arg\max_{\vec{\omega},d}\{\frac{1}{||\vec{\omega}||}\min_{n}[f_i(\vec{\omega}^T \vec{x_i} + d)]\}$$

which is the same as:

$$\arg\min_{\vec{\omega},d} \frac{1}{2}||\vec{\omega}||^2, \ subject\ to\ f_i(\vec{\omega}^T \vec{x_i} + d) \geqslant 1$$

Then we use Lagrange Multiplier:

$$L = \frac{1}{2}||\vec{\omega}||^2 - \sum_{n=1}^{N} a_n \times \{f_n(\vec{\omega}^T \vec{x}_n + d) - 1\}$$

The partial derivative $\frac{\partial L}{\partial \vec{\omega}} = 0$ and $\frac{\partial L}{\partial d} = 0$ are calculated and put the results back to the L equation. Then we could get the parameters $\vec{\omega}$ and d.

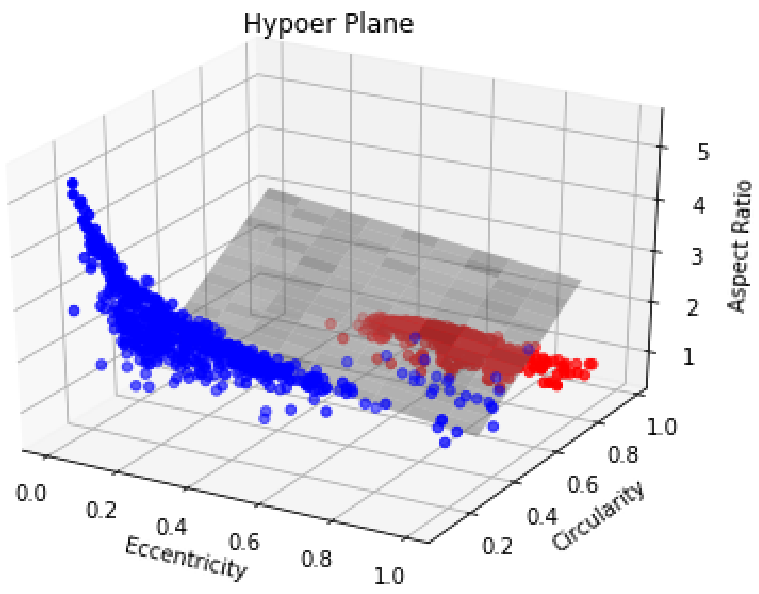


Figure 3.13: The 2d plane that separates the two categories. Its parameters are $\vec{\omega} = (-0.76151752, 4.011278, -1.08616378)$ and $d = -0.66255707$.

To classify a test point $\vec{u}$, the decision rule is:

$$If\ \vec{\omega}^T \cdot \vec{u} + b \geqslant 0,\ Then\ 'DOT'$$
$$If\ \vec{\omega}^T \cdot \vec{u} + b \leqslant 0,\ Then\ 'MORPHISM'$$

Then we use this trained classifier to classify each node in each hypothesis. After the classification phase, all the shapes in the hypotheses set are classified into wires, dots, and morphisms.

3.3.3 Morphism Orientation

According to the ZX-calculus rules, the morphisms can have four orientations. According to Figure 3.14, the four orientations include an original direction, a horizontal flipping direction, a vertical flipping direction and a horizontal-and-vertical flipping direction.

When we calculate the features of the shape, we calculate the centroid position and the bounding box of them. These two attributes are used for calculating the morphisms' orientation.

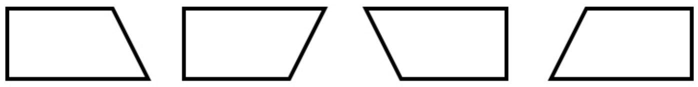


Figure 3.14: The four orientations of a trapezoid.

To calculate the orientation, we find out the farthest point on the bounding box from the centroid. Like the examples in Figure 3.15, we divide the space into four parts. The blue point is the centroid and the red point is the farthest point. The trapezoid's centroid point is placed on the origin coordinate so that we can calculate the sharp corner's direction according to the centroid.

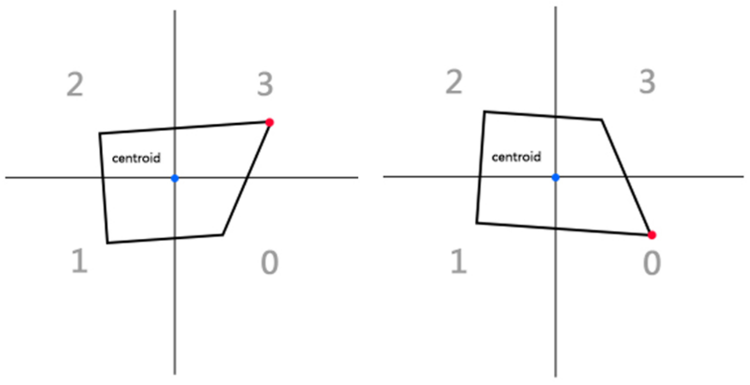


Figure 3.15: The method we use to determine the morphisms' orientation.

3.4 Connection

In each hypothesis, we have a set of wires and a set of nodes. We will connect the wires and nodes to make a diagram. Then these diagrams are given a score, and we choose the hypothesis with the highest score as the recognition result.

3.4.1 Connect and Score Rules

For one end of the wire, we iterate the node set and calculate the distance between the end position and each point position on the node contour. A threshold distance value is set, and only a distance smaller than the threshold will be recorded. After iterated all the nodes, we pick the node with the smallest distance in the history as the node to be connected with this wire and make this connection information as attributes of this wire and the connected node. We do this for each end of all the wires in the wire set.

After we've checked the connection, we iterate all the elements including wires and nodes in one hypothesis to check their connection attributes and give the score according to the rules below:

- Set the default $score = 1.0$
- If an element is connected to nothing, $score = score \times \alpha_1$
- If an element is connected to one another element, $score = score \times \alpha_2$
- If an element is connected to two or more other elements, $score = score \times \alpha_3$

The relationships of the α_i should be $\alpha_1 < 1.0 < \alpha_2 < \alpha_3$. These parameter are manually tuned by testing on the dataset.

3.4.2 Reject Wrong Segmentation

Our strategy is to use the connection rules to find out the most reasonable segmentation. Here we give some examples of diagram segments to show the rationality of the scoring rules.

Segment 1 is an example of a broken wire. No matter the complete wire is connected to a node or not (no matter we have path1 and path4 or not), the scoring rules always tend to group the strokes of the broken wire.

Segment 2 is an example of gathered wire ends (the ends of path2 and path3). Our rules will separate these ends instead of connecting the wires (path2 and path3).

Segment 3 is an example of a node with multiple strokes. The rules tend to group the strokes instead of leaving them separated.

We don't include the confidence of SVM classification in the score. The main reason is, there are a lot of other shapes that have similar features with circles and trapezoids. It is hard to find out some features that can rule out all the other shapes.

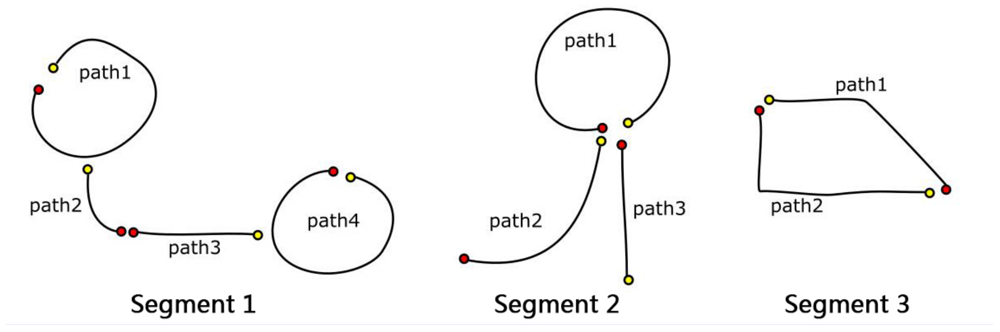


Figure 3.16: Three examples of diagram segments.

Example No.	Hypo No.	Switched-on Matches	Score
Segment 1	0	None	$1.05^4 = 1.2155$
	1	(path2,path3)	$1.05^2 * 1.2 = 1.323$
Segment 2	0000	None	$1.05^3 = 1.1576$
	0001	(path1,path2,path3)	0.8
	0010	(path2,path3)	$0.8^2 = 0.64$
	0100	(path1,path3)	$0.8^2 = 0.64$
	1000	(path1,path2)	$0.8^2 = 0.64$
Segment 3	0	None	$0.8^2 = 0.64$
	1	(path1,path2)	0.8

Table 3.1: The score of different hypotheses for each diagram segments using parameters $\alpha_1 = 0.8$, $\alpha_2 = 1.05$ and $\alpha_3 = 1.2$. Only valid hypotheses are listed.

So, in the classification stage, once the shape is a closed convex shape, we always consider it a node and give a result of ”dot” or ”morphism” even it’s a nonsense shape. And also, we won’t use the confidence value as the reference of the score because a nonsense shape is very likely to get a confidence as high as a well-drawn circle or trapezoid.

3.5 Correction

We also consider users’ correction behaviour in our algorithm. When the algorithm is embedded in a real-time program, the algorithm could generate hypotheses and produce a result each time the user draws a stroke on the canvas. We use edit modes to describe the user's intention of drawing a new stroke and add correction-related functions to the three previous stages.

3.5.1 Edit Modes

We define two edit modes to describe the intention of a new stroke: Replace or Enhance. Replace mode means that the latest stroke is to replace another stroke, and Enhance mode means that the latest stroke is to add a new stroke in the diagram.

We set a simple rule to classify the most recent stroke to these two modes: iterate all the previous paths and calculate the intersection between the previous path and the new path. If the new stroke doesn't intersect with any previous stroke, then it is entirely in Enhance mode. Otherwise, there is a possibility that it's in Replace mode. The more intersections between the two paths, the larger the possibility is. We choose the previous path which has most intersections with the new path as the one to be replaced.

3.5.2 Correct Rules

We classify a stroke to one of the two edit modes before the beginning of the segmentation stage. If it's Enhance mode, we do nothing in all the previous stages. If not, we'll record the replaced path and the intersection number, and apply some additional rules to segmentation, classification and connection stages.

- Segmentation
  We will generate two sets of hypotheses. Assume there were n paths on the canvas and the user draws a new stroke which is the $(n+1)$th path. The $Path_{n+1}$ is intersected with $Path_i$. We first generate a hypothesis set named $set_1(Path_1, Path_2, ..., Path_n, Path_{n+1})$ for the Enhance mode, which is without correction. Then we replace $Path_i$ with the $Path_{n+1}$ and generate another set of hypotheses named $set_2(Path_1, ...Path_{i-1}, Path_{i+1}, ..., Path_{n+1})$ for the Replace mode.

- Classification
  Classify the shapes in the two sets of hypotheses.

- Connection
  Connect the nodes and wires in the two sets of hypotheses. For set1, the initial score is 1.0. For set2, the initial score will be the same or larger than 1.0.

- Number of intersections = 1, then the $score = score \times \beta_1$.
- Number of intersections = 2, then the $score = score \times \beta_2$.
- Number of intersections = 3, then the $score = score \times \beta_3$.
- Number of intersections $\geqslant 4$, then the $score = score \times \beta_4$.

The relationships between the β_i should be $1.0 \leqslant \beta_1 < \beta_2 < \beta_3 < \beta_4$When the intersection number is large, all the hypotheses in set2 will have a much greater initial score than the hypotheses in set1.

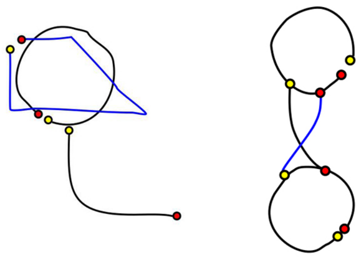


Figure 3.17: Two examples of diagram segmentation. The blue strokes are the latest stroke. In the left example, the blue stroke has four intersections with one of the previous stroke, so the hypothesis in the Replace mode will get a higher score. In the right example, there is only one intersection between the blue stroke and the previous stroke, so the hypothesis in the Enhance mode will get a higher score.

Example No.	Mode	Hypo No.	Score
1 (left example)	Enhance	0	$1.0 * 1.05^3 = 1.1576$
	Replace	0	$1.5 * 1.05^2 = 1.6538$
2 (right example)	Enhance	0	$1.0 * 1.2^4 = 2.0736$
	Replace	0	$1.05 * 1.05^2 * 1.2 = 1.3892$

Table 3.2: The score of the hypotheses for examples in Figure 3.17. For any two strokes in each example, their ends are not close enough to make the two strokes a pair. So there are only one hypothesis in each mode. In these examples, the connection parameters are $\alpha_1 = 0.8$, $\alpha_2 = 1.05$ and $\alpha_3 = 1.2$ and the correction parameters are $\beta_1 = 1.0$, $\beta_2 = 1.05$, $\beta_3 = 1.2$ and $\beta_4 = 1.5$.

In example 1, the latest stroke has four intersections with the previous stroke. If it is in Enhance mode, the initial score should be 1.0. There are three paths in the diagram, and each element is connected to one another elements. So the score is

1.1576. If the latest stroke is in Replace mode, the initial score will be 1.5. There are two paths in the diagram, and each element is connected to one another elements. So the score is 1.6538. The hypothesis in the Replace mode wins.

In example 2, the latest stroke only has one intersection with the previous stroke. If the new stroke is in Enhance mode, the initial score should be 1.0. There are four paths in the diagram, and each element is connected to two another elements. So the score is 2.0736. If the latest stroke is in Replace mode, the initial score will still be 1.0. There are three paths in the diagram. Each dot is connected to one wire, and the wire is connected to two nodes. So the score is 1.3892. The hypothesis in the Enhance mode wins.

Through these two examples, we can see such correct rules enable the users to make corrections to the diagram. However, the rules intend to reserve the intersections between wires when the number of intersections is small because this situation is very common in a ZX-calculus diagram.

3.6 Implementation and Output

We use Python code to implement the recognition algorithm. The input of the program is an SVG file, and the output would be structured data of the recognition results.

3.6.1 Program Structure

3.6.1.1 Classes

We create several classes to represent different forms of elements in different processing phases. They are paths in segmentation stage, shapes in classification stage, and nodes in connection stage. So there are five main classes used in the program:

- Path
  - Need a sequence of relative point positions to create a Path object.
  - Has the functions to calculate the absolute point positions and check intersection with other paths.
- Shape
  - Need a sequence of absolute point positions to create a Shape object.

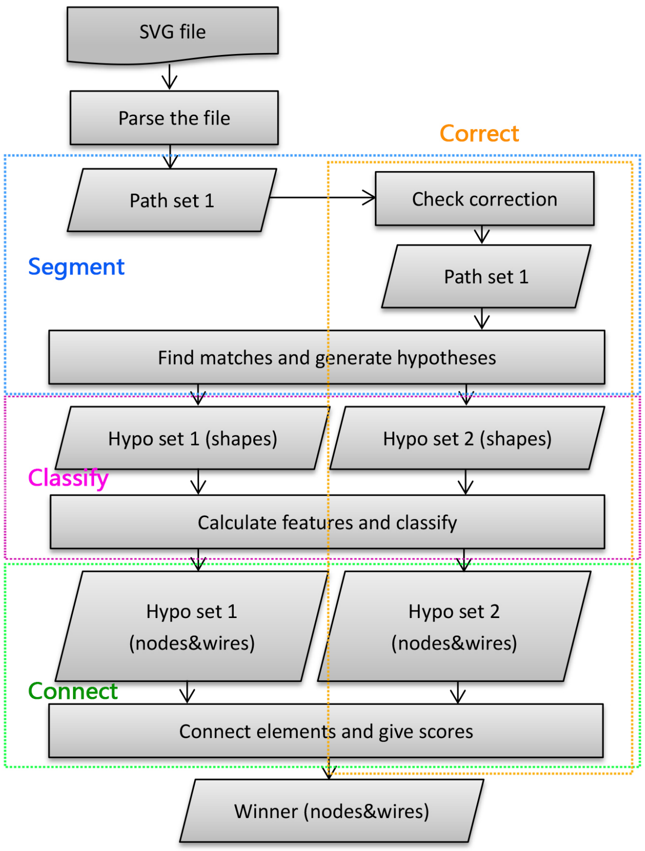


Figure 3.18: The structure of the program.

- Transform the point positions into polygons. Has the function to calculate shape features.

- Dot & Morphism & Wire

  - Need a sequence of absolute point positions and corresponding shape features to create a node.

  - Store the information of nodes' main features such as centre, orientation, and connection.

3.6.1.2 Functions

Then we create functions for the four recognition stages in three Python files: segment, classify and connect. The correction functions are distributed in these three files. The main functions of each file are:

- Segment

- Check the intersection between the latest path and all the previous paths in a path set. If there are one or more intersections, make a copy of the original path set and replace the old path with the latest path.
- Find matches in a path set.
- Generate all valid hypotheses for a path set.

• Classify
- Generate training data.
- Collect feature data from manually drawn SVG files and generated shapes.
- Train an SVM classifier.
- Classify shape objects in hypotheses to the wire, dot or morphism classes. Create corresponding node/wire objects and store them in the hypotheses.

• Connect
- Connect all the wires and nodes.
- Give scores to the hypotheses and find a winner.

There are also other functions for parsing the SVG files and drawing visualised outputs. We provide pesudo code for these functions in Appendix A.

3.6.2 Output Data

This algorithm gives recognition results in the form of structured data. When our recognition program is embedded in other programs, they could use the data to generate different kinds of visual outputs. The diagram below shows what information is included in the output data.

The winner hypothesis elected in the connection stage consists of a set of Dot objects, Morphism objects and Wire objects. We fetch these object attributes as the output data. If we still use Python to generate visual outputs, we can directly use these data to do the next step. If the algorithm is embedded in a program using other languages such like JSON, the output data will be transformed to a string to transfer the information.

Object	Attribute	Description
Dot	Centre	Centroid position
	Colour	Red/green, white/gray
	Connection	Wire objects' path points
Morphism	Centre	Centroid position
	Orientation	East/south/west/north
	Connection	Wire objects' path points
Wire	path points	All the points on the path
	connection	Node objects' centre, colour/orientation and connecting angle

Table 3.3

Chapter 4

Visualisation

As introduced in Chapter 3, The output we get from this algorithm is some structured data including recognised information. However, it is not convenient for both algorithm developers and users to evaluate the results. So we visualise the results in two ways: SVG images and LaTeX PDF. The SVG image is an unmodified feedback of the output data for the user to check their drawing, and the LaTeX command is a kind of application of this algorithm which can be directly used in papers. In this chapter, we briefly introduce the process we generate the visualised output and demonstrate some examples.

4.1 SVG Images

We generate SVG images simply with unmodified element objects and some XML commands. We have three kinds of shapes in the diagram which are dots, morphisms and wires. So we use circle and path elements in the XML:

- Dot

  Read the recognised dot's centre and define a radius. Put this information into a circle element to create a dot.

- Morphism

  Read the recognised dot's centroid and define edge lengths. Use the v-vertical lines and h-horizontal lines to create a right trapezoid.

- Wire

  Read the recognised wire's point list and pick 3-6 points uniformly. The positions of the two end points are replaced by its connecting nodes' (if there are connected nodes) edges. Then we can create a smooth wire by feeding the points' positions into curve commands.

The SVG images provide great convenience when developing the algorithm. Besides, it can be embedded into any program using this algorithm for the users to check whether they would get a right result with the current drawing. If the recognition is wrong, the user can make a correction on the canvas to correct it.

4.2 LaTeX PDF

As the ZX-calculus diagrams are for academic use, they are sometimes demonstrated on papers. So we also develop this function for the users who want to use ZX-calculus diagrams for demonstrating.

We use Tikz package to create dot and morphism templates, and also use it to draw the diagram. Before generating a neat ZX-calculus in LaTeX, the output data need to be reorganised first.

- Node

  Before drawing the nodes, we need to do following preprocessing to the recognised nodes:

  1. First, find the nearest two nodes $node_p$ and $node_q$ and make one of them as the origin coordinate, which means put $node_p$ at $(0,0)$. We use their distance $d_{min} = distance(node_p, node_q)$ as the unit edge length of the diagram.
  2. Create a grid with the unit grid length of d_{grid} (1.0 in our program). For each other node $node_i$, calculate its relative position (x_{rel}, y_{rel}) according to the centering node $node_p$. Then the $node_i$'s position in the grid should be $(round(\frac{x_{rel}}{d_{min}}), round(\frac{y_{rel}}{d_{min}}))$, where *round* is the function to round a float to a multiple of the d_{grid}.

  By the two steps above, we can know the positions of all recognised nodes on the grid. To draw these nodes in the latex, we just need to feed their types and grid positions into the drawing commands.

- Wire

  We define two types of wires in the LaTeX code. One is the basic wire, and another is the long wire. Both of them need to be created by an out-node, an in-node, an out-angle and an in-angle. The difference is that the long wires also include some inter-points. Inter-points are points between the start point and the end point of a wire. With these points, we can draw longer wires with any curvature we want. Again, we need to do following preprocessing before drawing the wires:

1. We first create another grid with a higher density (0.5 for the unit grid length in our program). For the wires connecting to nodes, the connected nodes become their in-node and out-node. For those wires whose ends connecting to nothing, we put the end points of these wires onto the wires' grid. Then the end points become their in-node or out-node.
2. The in-angle and out-angle are the connecting angles which could be retrieved directly from the output data. If the wire is connecting to a dot or connecting to nothing, we round the angle value to a multiple of 30.0 to make the diagram look tidy. If the wire is connecting to a morphism, it will always come in or come out the morphism vertically. When there is more than one wire connecting to a morphism, these wires will be placed evenly on its sides.

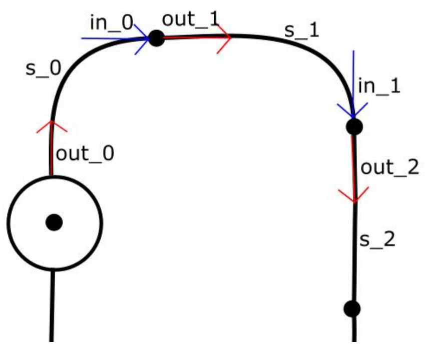


Figure 4.1: A part of a long wire.

3. If the wire's length is longer than $1.5 \times d_{min}$, it will be defined as a long wire. We pick inter-points from the wires' point list and calculate their relative positions in the wires' grid. The wire segments between each two inter-points also have their own out-angle and in-angle. For example, we have three wire segments in Figure 4.1 include s_0, s_1, and s_2. The blue arrows represent in-angles, and they are calculated by the relative positions between the two inter-points of the segment. The red arrows represent out-angles, and they (excluding out_0 which is the out-angle of the whole wire) are always in the opposite direction

to the in-angle of the previous segment, such like $out_1 = (in_0 + 180)\%360$. This setting can make the long wire go smoothly.

To draw a wire in LaTeX, we first feed the out-node, in-node, out-angle and in-angle information into the command, and then check whether the wire is a long wire. If yes, add the inter-points into the command.

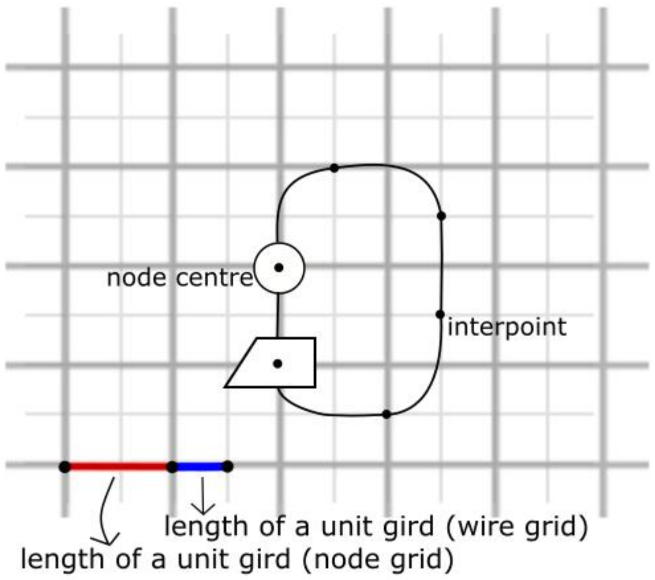

node centre
interpoint
length of a unit gird (wire grid)
length of a unit gird (node grid)

Figure 4.2: The diagram we draw in the LaTeX. The nodes are placed on the grids with the lower density, and the wire points including all the inter-points are placed on the grids with the higher density.

We show two examples of SVG visualisations and LaTeX visualisation. According to Figure 4.3 and Figure 4.4, both two visualising ways can correctly reconstruct the hand-drawn diagrams.

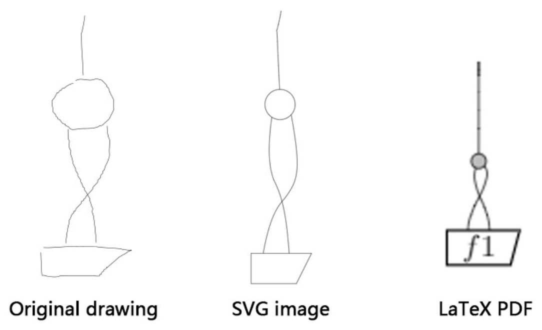


Figure 4.3: The first example of visualisation. The dot node uses default colour which could be changed in the code. The wires linked to a morphism are evenly arranged in LaTeX output.

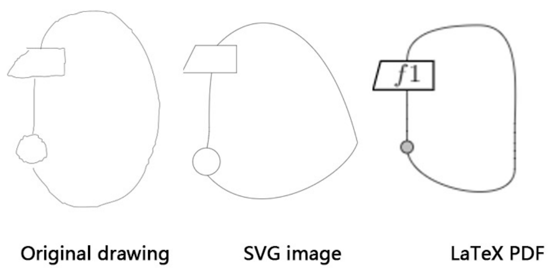


Figure 4.4: The second example of visualisation. In the LaTeX output, We add inter-points in a wire command to draw the long and bending wire in the diagram.

Chapter 5

Evaluation

In this chapter, we evaluate the recognition performance of the algorithm in chapter 3 and the output quality of visualisation in chapter 4. The algorithm evaluation is mainly based on the accuracy of shape classification and the accuracy of the whole hypothesis-based recognition system. It demonstrates the algorithm's recognition ability.

5.1 Shape Classification

In this section, we report the accuracy of classifying. The performance of classifying wires and nodes is based on the parameter setting, and the performance of dots and morphisms classification relies on the SVM classifier's ability. We will test them together with one dataset.

5.1.1 Dataset

We use around 2000 shapes including circles and right trapezoids as the training data for the SVM classifier. Two real users create 20% of these shapes and the rest 80% are generated by code. There are another about 300 manually created shapes for evaluating. One of the two users doesn't know the classifying rules in this algorithm at all.

We didn't ask the users to draw wires in the dataset because it is meaningless to test wires drawn by users who don't know the ZX-calculus rules. Or we can say it is ”dangerous” to test wires separately outside of the diagrams. There might be wires that look like nodes. If we want to know the real proportion of these wires and how our parameters work, we must ask those who know the ZX-calculus very well to draw the diagrams and then do the test. So we gave up this part in our evaluation.

	Parameter	Value	Description
Real hand-drawn shapes	Smoothness	20	The smoothing level of freehand strokes in Inkscape
	Snapping	0	The threshold distance that determines whether a stroke's end should snap another's
Code-generated shapes	Circle radius r	(60,80)	The semi-diameter of a circle
	Trapezoid edge lengths a,b,h	a$\in (40-70)$ b$\in (50-90)$ h$\in (30-80)$	The lengths of the top edge, the bottom edge and the left edge.
	Point number n	$(80-120)$	The number of points in a generated shape path
	Deformation ratio θ	0.03	The movement when deform a shape Circle: $(-0.03, 0.03) * r$ Trapezoid:$(-0.03, 0.03) * \sqrt{a^2/4 + h^2/4}$
SVM Classifier	Features	Eccentricity& Circularity& Aspect ratio	The three features described in chapter 3
	Kernel	Linear	Use the linear function as the kernel
	Loss	Hinge	Loss function
	Penalty	L2	The norm for the penalization
	Iteration	1000	The maximum iterations to run
	tol	1e-4	Tolerance for stopping criteria
Wire classifying	Open threshold	0.10	The threshold value of opencheck ratio for a shape element.
	Convex threhold	0.70	The threshold value of convexcheck ratio for a shape element.

Table 5.1: The settings of the classification part in the program.

All the settings and the parameters we use in the classifying evaluation are introduced in Table 5.1. As for the tools, we use Inkscape as the drawing board to create real freehand shapes, SVM package from Scikit-learn to train the classifier.

5.1.2 Metrics and Results

For wire and node classifying, we set the accuracy of classifying nodes as the evaluation metric, since there are no wires in our dataset.

$$Accuracy = \frac{Nodes}{All\ the\ shapes} = \frac{2}{325} = 99.38\%$$

For dot and morphism classifying, we set the precision, recall and the F-measure of the recognised result as the evaluation metrics.

		Actual	
		Dot	Morphism
Predicted	Dot	154	0
	Morphism	1	168

Table 5.2: Confusion matrix of the SVM recognised result.

$$Precision = \frac{\{relevant\ shapes\} \cap \{retrieved\ shapes\}}{\{retrieved\ shapes\}} == 100\%$$

$$Recall = \frac{\{relevant\ shapes\} \cap \{retrieved\ shapes\}}{\{relevant\ shapes\}} = 99.35\%$$

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall} = 99.67\%$$

5.1.3 Discussion

Our classification algorithm classifies two nodes into the wire class. In our parameter setting, if the distance between a stroke's two ends exceeds one-tenth of its parameter, the stroke will be defined as a wire. According to the wrongly classified stroke which we retrieved from the dataset in Figure 5.1, it's hard to tell whether it is more like a dot or a wire by our observation because the gap between the ends is huge. We conclude that our classifying rules for wires might be reasonable but too absolute. It is not sensible to define a sketch stroke simply by all-or-nothing rules because there is much uncertainty in the sketches. As we already establish a hypothesis-based system, generating hypotheses for uncertain wire classification may be a good solution to this problem.

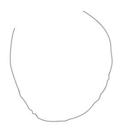


Figure 5.1: This shape is in the dot dataset but classified into the wire class by our classifying algorithm.

For dot and morphism classification, we conclude that the SVM is a very reliable classifier according to the results and it applies to our program. The main reason why it performs well is that circles and right trapezoids are relatively simple shapes with distinctly different features. That means the SVM should be a good choice for classifying diagram elements even when there are more kinds of shapes in the diagram. However, it has the drawback that we need to find the features to separate the categories and test them, which might be time-consuming and such features do not always exist.

We retrieve the wrongly recognised dot in the test images and find that the recognising failure is mainly caused by the poor quality of drawing (Figure5.2). We believe a human will classify it into the right class because we can see the curvature of this shape, while curvature is not valid as we choose a low smoothness of freehand strokes to reserve the sharp corners of the trapezoids. However, this is the drawback of the input images, not the classifier's. If there exist some input formats from which we can calculate the curvature and add it to the SVM, the classifier will perform better.

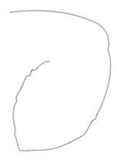


Figure 5.2: This shape is in the dot dataset but classified into the morphism class by our classifier.

5.2 Diagram Recognition

As the segmentation and connection are two interactive parts and they constitute the hypothesis-based scoring system, so we decide to test them together. So we report the accuracy of diagram recognition with correction and without correction in this section, and discuss the correction system and the overall algorithm design.

5.2.1 Dataset

We create 300 ZX-calculus diagrams using Inkscape by three real users. One of the three users doesn't know the rules in this algorithm. After testing the accuracy without correction, we add one more stroke on each of the correctly recognised diagrams to make a correction dataset. The correction stroke might be correcting nodes or wires.

	Parameter	Value	Description
Segmentation	Match distance	10	The maximum distance between two paths' ends that determines to group them
Connection	Score parameters α_i	$\alpha_1 = 0.8$ $\alpha_2 = 1.05$ α_3=1.2	The score of an element with 0/1/2 connections
Correction	Score parameters β_i	$\beta_1 = 1.0$ $\beta_2 = 1.1$ $\beta_3 = 1.2$ $\beta_4 = 1.5$	The score of the correction set with 1/2/3/more than 4 intersections between the last path and one previous path
Generated diagrams	Canvas size	A4	The size of the canvas we draw the diagram on
	Number of nodes	(2,7)	The number of nodes we draw in the dataset

Table 5.3: The settings of the whole diagram recognition process in the program. The SVG drawing settings is the same as the settings in classification evaluation.

5.2.2 Metrics and Results

We set the recognition accuracy as the metric for evaluating uncorrected diagrams. Only when the recognizer correctly recognises all the nodes and the connections, we define the diagram a correctly recognised diagram.

$$accuracy = \frac{correctly\ recognised\ diagrams}{all\ diagrams}$$

For evaluation of the correction, the metric is the percentage of successful corrections. We compare the re-recognised diagrams and the users' correct intention. If the result produced by the recognizer is exactly what the users want, we define it a successful correction.

	Accuracy / successful correction rate
uncorrected diagrams	94.33%
corrected diagrams	69.26%

Table 5.4: The evaluation results of the whole recognition system.

5.2.3 Discussion

The accuracy of the uncorrected recognition system is still over 90% but is much lower than the classification. We retrieve the wrongly recognised examples and discuss the reasons for the failures to see which aspects of the algorithm still needs improvement.

There are mainly two types of failure recognition of uncorrected diagrams. One is that multiple elements are drawn with one stroke and the recognizer considers they were only one element. We made a short interview asking the users who drew the dataset about their drawing habits, and it showed the users didn't tend to draw multiple elements in one stroke. So such failure should be caused by the snapping function of the drawing tool Inkscape. Although we closed the snapping function, the snapping still might happen when the user starts a stroke exactly on the end of another stroke, and combine two strokes into one. No matter it is the users' intention to use one stroke to draw two elements, or it's the mechanism of the drawing tool, this failure shows an apparent defect of our algorithm that it doesn't have the function of segmenting single strokes in the diagrams. If we can solve this problem, our algorithm can tolerate strange drawing behaviours and unwanted snapping, which will greatly increase the recognition accuracy.

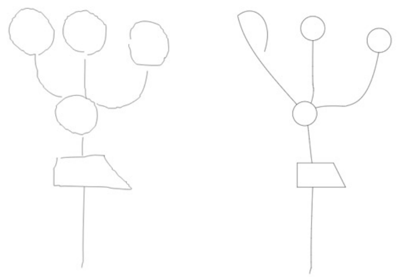


Figure 5.3: The example of failure recgonition which is caused by drawing multiple elements with one stroke.

Another kind of failure happens when the user uses four or more strokes to draw one element. Our algorithm only supports groups with three or fewer strokes, as it's unlikely that users use too many strokes to create a shape. We also don't recommend the users to do so because it will generate many hypotheses in the segmentation stage which need lots of computation. So the maximum number of stroke 3 is a reasonable setting with the current algorithm. We might increase the number of strokes in one group or even cancel the upper limit of this number in the future if we can improve the computation problem of the hypotheses generating.

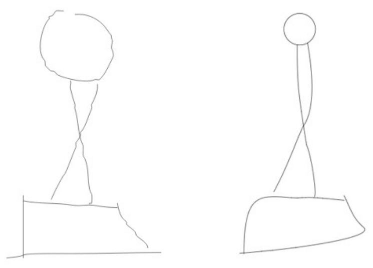


Figure 5.4: The example of failure recgonition which is caused by drawing an element with more than three strokes.

In the testing of corrected diagrams, the result is that the algorithm works well for node correction but not for wires. The failure happens when the users draw a stroke with zero or one intersection with the target stroke. Based on our scoring rules, it's difficult to know whether the user wants to add a wire or correct a wire when there is only one or less intersection, especially one intersection between wires is common in ZX-calculus diagrams. In fact, sometimes it's even hard for a human to tell what is the user's real intention. The current solutions for this problem could be telling the users our rules of correction or adding some ZX-calculus connection knowledge to the scoring rules.

To conclude, our hypothesis-based system is a good recognizer for the ZX-calculus diagrams. It is especially applicable for sketch recognising because the hypotheses can solve the uncertainty problem of the drawing. However, it still might fall when receiving unusual behaviours or poor drawing from the users.

Chapter 6

Conclusion

6.1 Summary

In this thesis, we confirm the hypothesis that the recognition algorithm for ZX-calculus diagrams exists.

The algorithm is divided into four stages which are segmentation, classification, connection, and correction. In the first three stages, we segment the diagram sketch into elements, classify them and then reconstruct the diagram by connecting these elements. The correction is embedded in these three stages to help users correct strokes drawn by mistakes. The whole process is a hypothesis-based system.

We evaluate the main component classifier and the entire system by the dataset created by ourselves. The results show that our algorithm is a feasible solution, but there are still some small defects need improvement in the future.

6.2 Future Work

According to the discussion in Chapter 5, there are mainly three defects in the algorithm that we can improve in future work. As time is limited, we leave these three problems for future work. But for each problem, we propose one or two possible solutions.

The first issue is the algorithm might classify nodes into the wire class. This can be solved by generating hypotheses for uncertain classifications. In our original algorithm design, a stroke will be defined as a wire if its open ratio is over 0.1 or its convex ratio is lower than 0.70. We propose to use range values and the probability instead of the absolute criterion. For example, we set the ranges for the two ratios at

(0.05,0.15) and (0.55,0.85). Then there is 50% probability for the stroke which has the open ratio of 0.1 and the convex ratio of 0.85 to become a wire or a node. When the open ratio increases or the convex ratio decreases, the probability of being a node goes down. Otherwise, it is more likely to be a node. We can generate hypotheses and score them with the probability. When both two ratios are out of this range, we do not generate hypotheses.

The second is the algorithm fails to recognise multiple elements with one stroke. One possible solution that we will try in the future is to segment each single stroke and generate more hypotheses. Assuming there are n points in one stroke, we could check the distance between each two points on one stroke. If two points p_i and p_j are very close to each other and the path from p_i to p_j composes a convex shape, we will generate one more hypothesis in which this stroke is segmented into $Path_1(p_1,...,p_i)$, $Path_2(p_i,...,p_j)$ and $Path_3(p_j,...,p_n)$. Then we might be able to find out some of the situations that the users use one stroke to draw more than one element.

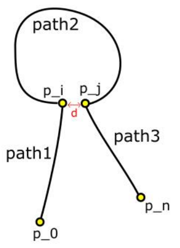


Figure 6.1: The example of drawing one nodes and two wires with one stroke.

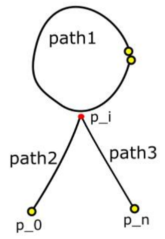


Figure 6.2: The example of drawing multiple wires with one stroke.

However, the solution above can only solve the situation in Figure 6.1. If the user draws multiple wires with one stroke, it won't segment it. Another possible solution is to find abruptly changing tangent along the stroke. In Figure 6.2, we will chop the $Path(p_0, p_n)$ into $Path2(p_0, p_i)$ and $Path(p_i, p_n)$ as there is a sharp corner at p_i. But the strokes drawn by a free hand pen with a low smoothness are jagged. (The low smoothness is to reserve precise shape of the elements.) If we want to use this solution, we must find out the way to preprocess the strokes to make them smooth but still keep the corners.

The third defect is the algorithm can't tell whether the users want to add a new stroke or correct the old stroke when there is only one intersection between the latest stroke $Path_n$ and the old stroke $Path_i$. To solve this problem, we plan to add the knowledge of ZX-calculus into the scoring rules. When a user adds a new stroke and we determine that there is a possibility that the user is trying to correct the diagram, we will assess the two wire associations in Enhance mode and Replace mode including the new stroke. Assume that $Path_p$ and $Path_q$ have the same input node or output node as $Path_n$ or $Path_i$. We'll check $Association_2(Path_p, Path_q, Path_n)$ (where $Path_i$ is corrected) and $Association_1(Path_p, Path_q, Path_i, Path_n)$ (where $Path_n$ is the new stroke). If a wire association is common in the ZX-calculus, it will receive a high score over 1.0. Otherwise, it's score will be lower than 1.0. This method might not be able to get the users' intentions right all the time since it's even confusing to human eyes. But it should be able to increase the accuracy rate for wire corrections significantly.

In conclusion, the hypothesis mechanism and the ZX-calculus knowledge are essential elements of our algorithm. By improving the functions in these two aspects, our algorithm could overcome ambiguities, increase the tolerance for unconventional users' behaviours, and decrease the error rate of correction judgment.

Appendix A

Pseudo Code

In this appendix, we provide the Pseudo Code of most important functions in each stage of the algorithm. For full code and examples, please see https://github.com/emmacaort/freehand-zxcalculus-recognition.

A.1 Segmentation

Algorithm 1: Generate hypotheses with matches found in the path list

```
Generate Hypotheses ($pathlist, train$);
if $train = False$ then
    $matchlist \leftarrow FindMatches(pathlist)$
    for $hypothesis \leftarrow$ YieldHypothesis(pathlist,matchlist) do
        $TransToShapes(hypothesis)$
        $hypotheses \leftarrow AddToHypotheses(hypothesis)$
    end
    return $hypotheses$
else
    $TransToShapes(pathlist)$
    $hypotheses \leftarrow AddToHypotheses(pathlist)$
    return $hypotheses$
end
```

Algorithm 2: Replace the old stroke if there are intersections

```
1  Replace Stroke by correction ($pathlist$);
2  $lastpath \leftarrow FindLastPath(pathlist)$
3  $max\_intersection \leftarrow 0$
4  forall $path \in pathlist$ do
5      $intersection \leftarrow CountIntersection(lastpath, path)$
6      if $intersection > max\_intersection$ then
7          $max_intersection, replace\_path \leftarrow intersection, path$
8      end
9  end
10 if $max_intersection > 0$ then
11     $pathlist \leftarrow RemoveStroke(pathlist, replace_path)$
12 end
13 return $pathlist$
```

A.2 Classification

Algorithm 3: Train the SVM with existing files and generated shapes

```
1  Train the SVM ($trainfile, dot\_n, mor\_n$);
2  $generated\_nodes \leftarrow GenerateNodes(dot\_n, mor\_n)$
3  forall $node \in (trainfile, generated\_nodes)$ do
4      $feature2 \leftarrow ExtractFeature(node)$
5      $label2 \leftarrow GetLabels(node)$
6  end
7  $classifier \leftarrow trainSVM(feature, label)$
8  return $classifier$
```

Algorithm 4: Classify a shape and create a wire or node object

```
1  Classify a shape ($shape, classifier$);
2  $concave \leftarrow CheckConcave(shape)$
3  $open \leftarrow CheckOpen(shape)$
4  if $convex$ and $open$ then
5      $wire \leftarrow CreateElement(shape, 'wire')$
6      return $wire$
7  else
8      $nodetype \leftarrow classifier(shape)$
9      $node \leftarrow CreateElement(shape, nodetype)$
10     return $node$
11 end
```

A.3 Connection

Algorithm 5: Give the score to a hypothesis

```
ScoreHypothesis ($hypothesis, parameters$);
$correct\_param, connect\_param \leftarrow parameters$
$score \leftarrow 1.0$
$intersection\_n \leftarrow Getintersection(hypothesis)$
$score \leftarrow CorrectScore(score, correct\_param, intersection\_n)$
forall $element \in hypothesis$ do
    $score \leftarrow ConnectScore(score, connect\_param, element)$
end
return $score$
```

Algorithm 6: Give the score to a hypothesis

```
FindWinner ($enhance\_set, replace\_set, parameters$);
$max\_score \leftarrow 0.0$
forall $hypothesis \in (enhance\_set, replace\_set)$ do
    $score \leftarrow ScoreHypothesis(hypothesis, parameters)$
    if $score > max\_score$ then
        $winner \leftarrow hypothesis$
    end
    return $winner$
end
return $hypothesis$
```

A.4 Visualisation

Algorithm 7: Get the centre point and unit distance of the diagram for the grid

```
1  CreateGrid $(nodelist)$;
2  $centroids \leftarrow GetCentroids$
3  $min\_d \leftarrow a\ large\ number$
4  for $(node1, node2) \leftarrow YieldNodePairs(nodelist)$ do
5      $d \leftarrow distance(node1, node2)$
6      if $d < min\_d$ then
7          $min\_d \leftarrow d$
8          $centre \leftarrow node1$
9      end
10 end
11 return $centre, min\_d$
```

Algorithm 8: Draw a node element in LaTeX

```
1  DrawNode $(node, centre, min\_d, unit\_d)$;
2  $attributes \leftarrow GetAttributes(node, centre, min\_d, unit\_d)$
3  $command \Leftarrow ComposeNodeCommand(attributes)$
4  return $command$
```

Algorithm 9: Draw a wire element in LaTeX

```
DrawWire ($wire, centre, min\_d, unit\_d$);
forall $end \in GetEnds(wire)$ do
    $connection \leftarrow GetConnection(end)$
    if $connection \neq None$ then
        $connect\_point \leftarrow GetNodeCentre(connection)$
    else
        $connect\_point \leftarrow end$
    end
    $end\_attr \leftarrow CalculateConnectAttr(connect\_point, centre, min\_d, unit\_d)$
    $ends\_attr \leftarrow AddToCommandAttr(end\_attr, ends\_attr)$
end
if $CheckLongWire(wire)=True$ then
    $interpoints \leftarrow GetInterpoints(wire)$
    $inter\_attr \leftarrow$
      $CalculateInterAttr(interpoints, ends\_attr, centre, min\_d, unit\_d)$
end
$command \Leftarrow ComposeWireCommand(ends\_attr, inter\_attr)$
return $command$
```

Bibliography

Angadi, M. and Lakshman Naika, R. (2014). Handwritten circuit schematic detection and simulation using computer vision approach. *International Journal of Computer Science and Mobile Computing*, 3(6):754–761.

Auria, L. and Moro, R. A. (2008). Support vector machines (svm) as a technique for solvency analysis.

Awal, A.-M., Mouchère, H., and Viard-Gaudin, C. (2014). A global learning approach for an online handwritten mathematical expression recognition system. *Pattern Recognition Letters*, 35:68–77.

Calhoun, C., Stahovich, T. F., Kurtoglu, T., and Kara, L. B. (2002). Recognizing multi-stroke symbols. In *AAAI Spring Symposium on Sketch Understanding*, pages 15–23.

Cawley, G. C. and Talbot, N. L. (2010). On over-fitting in model selection and subsequent selection bias in performance evaluation. *Journal of Machine Learning Research*, 11(Jul):2079–2107.

Coecke, B. and Duncan, R. (2011). Interacting quantum observables: categorical algebra and diagrammatics. *New Journal of Physics*, 13(4):043016.

de Beaudrap, N. and Horsman, D. (2017). The zx calculus is a language for surface code lattice surgery. *arXiv preprint arXiv:1704.08670*.

Dreiseitl, S. and Ohno-Machado, L. (2002). Logistic regression and artificial neural network classification models: a methodology review. *Journal of biomedical informatics*, 35(5):352–359.

Ge, S. S., Hang, C. C., Lee, T. H., and Zhang, T. (2013). *Stable adaptive neural network control*, volume 13. Springer Science & Business Media.

Gennari, L., Kara, L. B., Stahovich, T. F., and Shimada, K. (2005). Combining geometry and domain knowledge to interpret hand-drawn diagrams. *Computers & Graphics*, 29(4):547–562.

Guru, D., Sharath, Y., and Manjunath, S. (2010). Texture features and knn in classification of flower images. *IJCA, Special Issue on RTIPPR (1)*, pages 21–29.

Hammond, T. A., Logsdon, D., Paulson, B., Johnston, J., Peschel, J. M., Wolin, A., and Taele, P. (2010). A sketch recognition system for recognizing free-hand course of action diagrams. In *IAAI*.

Kissinger, A. (2012). Pictures of processes: automated graph rewriting for monoidal categories and applications to quantum computing. *arXiv preprint arXiv:1203.0202*.

Li, Y., Song, Y.-Z., and Gong, S. (2013). Sketch recognition by ensemble matching of structured features. In *BMVC*, volume 1, page 2.

Montero, R. S. and Bribiesca, E. (2009). State of the art of compactness and circularity measures. In *International mathematical forum*, volume 4, pages 1305–1335.

Ouyang, T. and Davis, R. (2009a). Learning from neighboring strokes: Combining appearance and context for multi-domain sketch recognition. In *Advances in Neural Information Processing Systems*, pages 1401–1409.

Ouyang, T. Y. and Davis, R. (2007). Recognition of hand drawn chemical diagrams. In *AAAI*, volume 7, pages 846–851.

Ouyang, T. Y. and Davis, R. (2009b). A visual approach to sketched symbol recognition. In *IJCAI*, volume 9, pages 1463–1468.

Ouyang, T. Y. and Davis, R. (2011). Chemink: a natural real-time recognition system for chemical drawings. In *Proceedings of the 16th international conference on Intelligent user interfaces*, pages 267–276. ACM.

Wettschereck, D., Aha, D. W., and Mohri, T. (1997). A review and empirical evaluation of feature weighting methods for a class of lazy learning algorithms. In *Lazy learning*, pages 273–314. Springer.

Wu, J., Wang, C., Zhang, L., and Rui, Y. (2014). Sketch recognition with natural correction and editing. In *AAAI*, pages 951–957.