Handwritten Mathematical Expression Recognition using Graph Edit Distance

Marco Benelli
marco.benelli@stud.unifi.it

Luca Bindini
luca.bindini@stud.unifi.it

February 15, 2021

Abstract

Unlike handwritten characters recognition, handwritten mathematical expressions recognition (HMER) is an area yet to be explored and which can provide useful results for research purposes. This article proposes an approach to the recognition of mathematical symbols in order to identify all of the mathematical expressions in which they are present. A description of the techniques and technologies used will initially be provided and finally we will analyze the implementation details commenting on the results obtained in terms of performances.

1 Introduction

In this paper we will describe a pipeline for the recognition of symbols in mathematical expressions thus going into the field of HMER (handwritten mathematical expressions recognition). This pipeline is inspired from the paper *Keyword spotting in historical handwritten documents based on graph matching* [1]

The mathematical expressions used in this work are taken from the CROHME (Competition on Recognition of Online Handwritten Mathematical Expressions) dataset where there are thousands of handwritten mathematical expressions encoded in InkML but which will be converted into .png images to increase the scalability of the project, having the possibility to add other mathematical expressions from external sources (e.g. handwritten images).

The symbol whose presence we want to look for in the dataset will be passed to the program in the form of an image that will undergo an initial preprocessing, the same the mathematical expressions went through. This image preprocessing process, better described later in this article, essentially consists of image scaling, resizing and binarization.

Both the mathematical expressions and the "query symbol" are transformed into graphs and, through algorithms that perform the Graph Edit Distance (GED), the similarity of the graph representing the mathematical symbol from the query is evaluated with each symbol present in each expression of those in the dataset.

The creation of the graph starting from the image and GED are undoubtedly crucial points of the entire pipeline, and it is precisely for this reason that several variants have been created both for the representation in graph form (grid-approach and polygon-approach) and with regard to GED (based on angles or based on distance).

Finally, the performances were evaluated by using precision and recall as metrics and comparing them in explanatory graphs in the various types of approach.

1.1 Utilized tools

The whole work has been developed using the Python 3.8 language on PyCharm platform using some external libraries useful for implementation purposes.

- **scikit-image**[2] is a collection of image processing algorithms mainly used in the image preprocessing phase.

- **networkx**[3] is a Python library for the creation, manipulation and study of the structure, dynamics and functions of complex networks or in our case of graphs representing symbols. This library was useful both for the representation of the graph itself and for GED phase.

- **matplotlib**[4] is a Python library for creating static, animated and interactive plots. This library was useful for creating precision recall charts for performance evaluation.

- **numpy**[5] is a Python library useful for basic operations on multi-dimensional arrays and matrices. It has been used throughout the entire project.

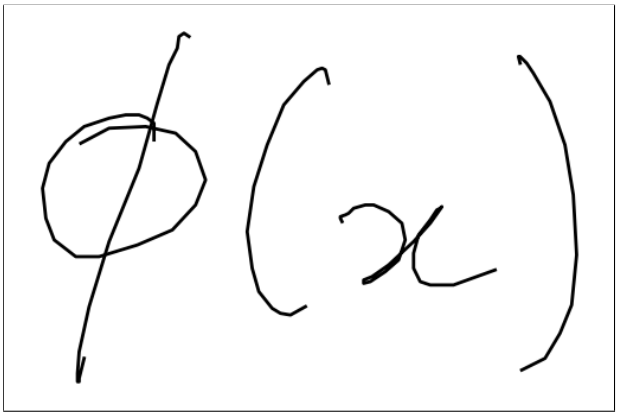

Figure 1: The expression after being converted to png.

2 Graph-based expression representation

As we said, our system uses graphs to represent the mathematical expressions. This means that each expression corresponds to an undirected graph, and each of its connected components corresponds to a symbol of the mathematical expression. Obviously the symbol segmentation done in this way will not be perfect, since some characters are composed of more pen strokes, like for instance the $\leq$ symbol. We will see how to get around this issue in section 3.

To get the graph representation, we must follow some basic steps. First we should have a phase of preprocessing in which artifacts like noisy backgrounds are eliminated and a lot of useless information is removed from the expression. Secondly, we should extract the actual graph, which can be done in multiple ways. This is one of the most delicate parts of the process, as there are many ways in which we could generate a graph from a ME. Therefore, we should take great care to make sure that the chosen representation is the right one.

2.1 Preprocessing

For the preprocessing step and all the others that follow it, the expressions have to be in image format. This will most likely be the case if you are using scans of documents, but on some occasions, like when using digital pens, you will be working with inkml files. Inkml files are essentially xml files with information about pen strokes as lists of 2D coordinates. These files can be easily converted into images using available libraries.

After the inkml files have been converted into png (meaning we obtained something like figure 1), the preprocessing phase includes many small steps to make the images have some fixed characteristics.

The first step involves proportionally resizing all of the images to the same height. We have to choose height and not width because text usually extends horizontally and not vertically. This means that resizing to the same height will make the characters approximately all of the same dimensions irregardless of the expression they come from.

As a second step, the images are binarized, meaning that from grey scale images, they become black and white image. This is done because a lot of the following processing only works on binary images. The way we got the black and white images was by simply choosing a threshold and considering as black the pixels below it, and as white the ones above it.

After we got the binarized image, we can start creating a graph. There are many algorithms to do this, we have chosen two: the first uses a grid-based

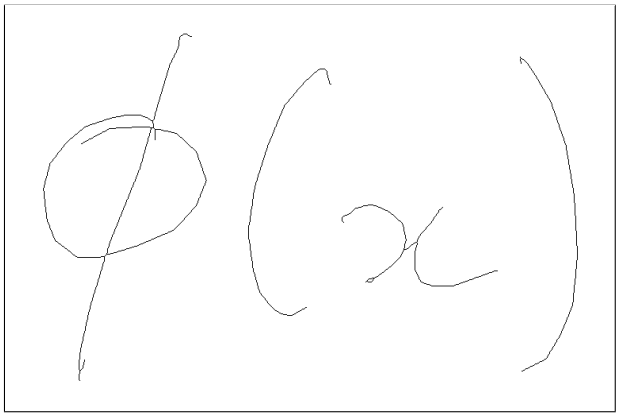

Figure 2: The skeleton of the image.

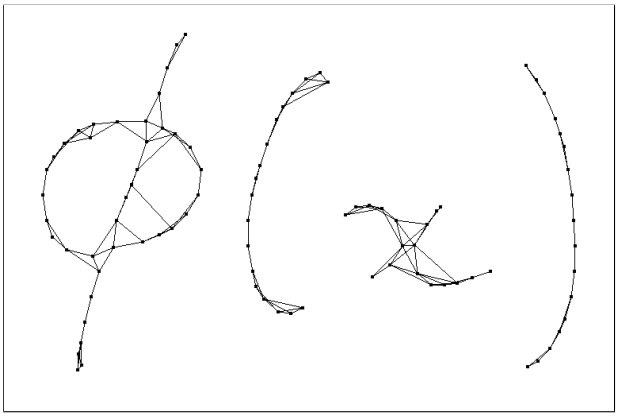

Figure 3: The graph obtained using the grid-based method with edges across adjacent nodes.

segmentation of the image, while the second one is polygon-based. Let's look at both of them in detail.

2.2 Grid

The grid method starts from the image's skeleton using *scikit-image*[2]. The skeleton of an image is a version of it in which all the lines are 1 pixel wide, like the one in figure 2.

After we have obtained the skeleton, as the name of this method implies, we have to divide it into cells all of the same dimensions. Each of these cells will correspond to a node of the graph if it contains a black pixel and this is how the nodes of the graph are defined. To define the edges we have a few choices that we'll explore next.

2.2.1 Edge to adjacent

The first and simplest criterion we can use to tell whether two nodes should share an edge or not is to see if they belong to adjacent cells (meaning that the square cells share a side). The resulting graph is shown in figure 3.

It is clear from this definition that this method is too simple, as some cells could be adjacent but may contain separate symbols. To avoid this from happening we would have to make the cells really small, but that would create a lot of nodes for a single character. This is a bad thing because the bigger the graphs are, the harder it will become to spot the similarities between graphs representing the same symbol.

2.2.2 Edge to connected

To avoid the problems explained above, we should use another criterion for determining edges. With this method we connect adjacent grid cells only if there is a line going through the edge they share. This results in the image in figure

This is considerably better than the previous method because it let's us choose a larger cell size, meaning we can reduce the number of nodes of the graph, while still maintaining a graph that is recognizable.

2.3 Polygon

There are a few problems with the grid-based method that we haven't touched on. As always, it uses too many nodes, even when using the better way of generating edges. Specifically, it's possible to represent a segment with many connected points on a single line instead of just two. The reason is that any edge can only connect two adjacent cells, because of how we generate the graph.

To avoid this issue, we have to use a totally different method. Instead of taking the skeleton of the binarized image, we should take its contours (the lines

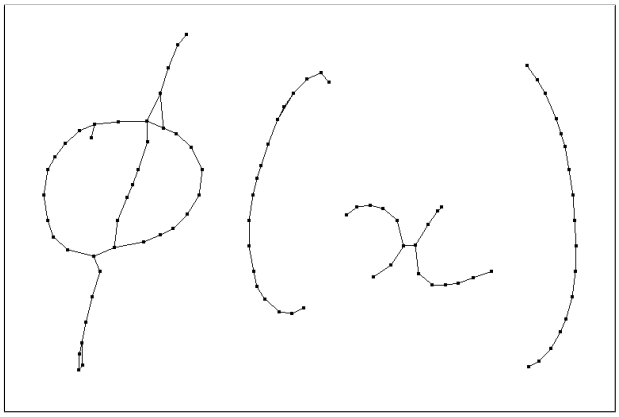

Figure 4: The graph obtained using the grid-based method with edges only across connected cells.

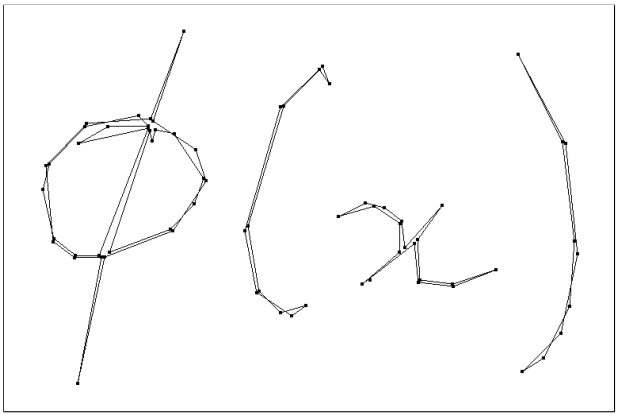

Figure 5: The graph obtained using the polygon-based method.

that separate the white area from the black area) and approximate them with polygons. This is done using the implementation of the Douglas-Peucker algorithm found in *scikit-image*[2] for the polygonal approximation.

As we can see in figure 5, this method proves to be the one that generates the smallest amount of nodes, and thus the one that will achieve the best performance.

2.4 Labels

Having chosen the contour-based method, we now face the problem of graph matching: if every symbol is represented by a cycle graph, the only thing that matters is the number of nodes and nothing else. This is the reason why we have to talk about labels before concerning ourselves with graph matching.

A labeled graph is simply a graph in which every node and/or edge has a label. These labels will be used when calculating the distance between two graphs in section 3. We will use two methods for labeling the graph.

The first approach relies on labeling each node with its 2D spatial coordinates and leaving the edges unlabeled. The second one involves labeling the nodes using the angle formed by the two edges meeting in each node. In this second method, the edges will be labeled with the measures of their lengths. We'll look at these methods in greater detail in section 3.

2.5 Normalization

Labeling all of the nodes with their position in the original image is obviously not going to be ideal when comparing the symbols with one another. The reason is that we want to consider the relative position of the nodes to each other, not their absolute positions in the expression they are taken from.

So, to normalize the graphs, we translate them and scale them down proportionately so that all of the coordinates are in the range from 0 to 1. Additionally, we also make sure that for one cardinal direction there is a node with 0 as the corresponding coordinate and another one with 1. This is to make sure that we won't scale down too much, but just enough to make the graph fit in a 1×1 square.

3 Graph Edit Distance

The Graph Edit Distance (GED) is a measure of similarity (or dissimilarity) between two graphs, used in this work to evaluate the "distance" between the graph of the searched symbol and the graph of each symbol present in the various expressions of the dataset. A cost is associated with each operation of

insertion, deletion and substitution of a node or an arc of the graph.

GED between the two graphs g_1 and g_2, which respectively represent the symbol used as a query and the symbols present in mathematical expressions, is defined as:

$$GED(g_1, g_2) = \min_{(e_1 \dots e_k) \in P(g_1, g_2)} \sum_{i=1}^{k} c(e_i)$$

where $P(g_1, g_2)$ denotes the set of edit paths transforming g_1 into g_2 and $c(e) \geq 0$ is the cost of each graph edit operation e.

Two different approaches were used to calculate the GED, providing the nodes of the graph with information on their position or information on the angle formed by the two sections adjacent to it. In both approaches the arcs have instead information regarding their length.

3.1 Angle

In this approach, the nodes of the graph have information regarding the angle between the two adjacent arcs. The information is useful to define the cost for the operations of insertion, deletion and substitution of the nodes for GED, in particular the weight of each node is $\sin(\alpha/2)$, where α is the angle between two arcs.

The cost of replacing a given node is the absolute value of the difference of node weights. The costs for deletion and insertion of a node depend directly on the node's weight itself.

3.2 Position

In this approach the graph nodes have information regarding their position, normalization was used to standardize the position of the nodes as described in section 2. This information was used to define the various costs of inserting, deleting and substituting nodes.

The cost of replacing a given node is the euclidean distance between nodes. The costs for deletion and insertion of a node in this case is constant.

4 Experimental evaluations

As with every proposed approach, we have to validate it through a series of tests. In this section, we will measure the performance of this method. We'll look at how the software performs when looking for a particular symbol in a set of expression.

4.1 Dataset

The first thing we need is a database of labeled handwritten mathematical expressions. The best ones come from *CROHME*, a Competition on Recognition of Online Handwritten Mathematical Expressions that takes place every year. Specifically, we'll be using a subset of the database from the 2011 edition.

The dataset consists of a series of inkml files baring not only the information for tracing the pen strokes, but also the labels associated with each pen strokes. These labels mark the ground truth (expressed in LaTeX form) corresponding to each trace or group of traces.

4.2 Query extraction

For the experiments we cannot use just a single query and look at how the program performs when looking for it, because that wouldn't be indicative of the program's true performance. What we should do is measure the performance across multiple queries and take the average. To do this, however, we need a number of queries that would be cumbersome to obtain manually.

To isolate a decent number of queries in an automatic way, we have to once again resort to the labeling present in the inkml files. Since the traces are all labeled, it's not hard to create a new inkml file containing only the traces relevant to a particular symbol.

One thing we want to avoid is querying for the same symbol multiple times. To keep this from happening we must keep track of the extracted symbols and avoid extracting the same ones more than once.

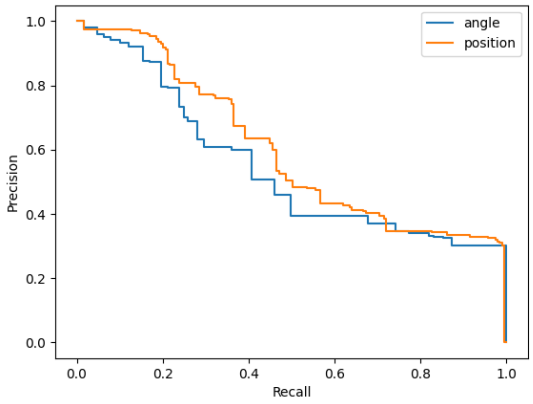

Figure 6: The precision-recall curves for both methods when looking at the averages for 43 different query symbols.

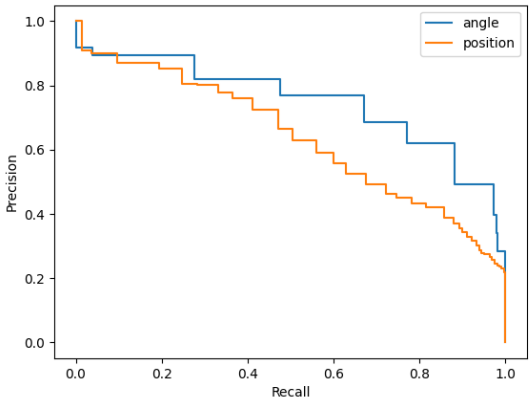

Figure 7: The precision-recall curves for both methods when querying for the symbol '('.

4.3 Precision recall

A common way to measure the performance is to look at the values of precision and recall. However, instead of looking at the absolute values, we should look at how they vary with respect to the GED threshold. The GED threshold is the value that determines whether two graphs correspond to the same symbol: if the GED is below the threshold, they are the same symbol, otherwise they are not.

The version of the precision-recall curves we'll take a look at is the interpolated one. Also, we'll plot the curve for both the angle method and the position method.

The results are the ones in figure 6. Those results might seem disappointing and that's because they are averages of many query symbols, including the ones for which the approach fairs worse. If we take a look at the results when querying for the symbol '(' in figure 7, we see big improvements.

5 Conclusions

As we have seen from the experimental results obtained, the two approaches, angle and position, obtain comparable performances in terms of precision-recall.

When looking for generic symbols, as can be seen from the figure 6, the performances are good but not optimal as this method does not work well with symbols composed of multiple pen strokes (e.g. less equal). If, on the other hand, we look for a less complex symbol (e.g. open bracket) we are able to obtain much better performances in terms of precision-recall.

References

[1] Michael Stauffer, Andreas Fischer, and Kaspar Riesen. Keyword spotting in historical handwritten documents based on graph matching. *Pattern Recognition*, 81:240 – 253, 2018.

[2] Stéfan van der Walt, Johannes L. Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, Tony Yu, and the scikit-image contributors. scikit-image: image processing in Python. *PeerJ*, 2:e453, 6 2014.

[3] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics,

and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[4] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[5] Charles R. Harris, K. Jarrod Millman, St'efan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fern'andez del R'ıo, Mark Wiebe, Pearu Peterson, Pierre G'erard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.