D.I.A.G.R.A.M.: Development of Image Analysis for Graph Recognition And Modeling

Filippo Garagnani, Saverio Napolitano, Nicola Ricciardi
'Computer Vision and Cognitive System' course
Università di Modena e Reggio Emilia

***Abstract*—This report describes D.I.A.G.R.A.M., a system for diagram recognition and generation. We present its architecture, the logic behind deterministic algorithms and the training of its deep learning components. The project aims to transform visual diagram input into structured representations.**

I. Introduction

Diagrams are crucial in education, documentation, and many other fields. Developing a system that automatically understands and generates diagrams may reveal helpful for having high-quality, easily editable representations. This, however, poses challenges involving classification, shape detection, structure interpretation and symbolic representation. Our project proposes a modular architecture to tackle these tasks.

II. System Architecture

Figure 1 illustrates the full pipeline, composed of four main modules: the Classifier, Extractor, Transducer, and Compiler. Each module is designed to process and transform the handwritten diagram image progressively toward a structured output.

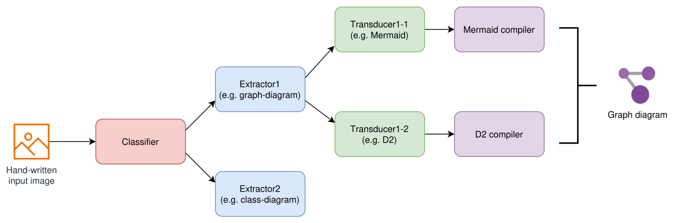


Fig. 1: Overview of the D.I.A.G.R.A.M. system architecture.

The Classifier is able to recognize the category of the hand-written diagram that the user submitted as input. This is necessary in order for the system to know to which Extractor pass the data. This module is able to apply object detection and semantic recognition to represent the diagram in a unified way. After that, the representation is sent to the Transducer, which converts the data in a Markup language of choice. The Markup language content is therefore sent to the Compiler, thus generating the high-quality and editable diagram in .png format.

A. Classifier Module

The Classifier is the first component of the pipeline. It is able to tell to which category a given image of an handwritten diagram belongs to. This is necessary in order to later know the Extractors that can process the input image.

B. Extractor Module

The Extractor is the key component of the D.I.A.G.R.A.M. system, able to transform the raw diagram image into a structured, category-agnostic representation. The Extractor recognizes the diagram's components, such as shapes, lines, and text, through an object detection network, and organizes them into a unified format. This representation serves as the foundation for the subsequent Transducer module, which converts the structured data into a domain-specific markup language.

C. Transducer Module

The Transducer is responsible for converting the unified, agnostic representation of a diagram into a domain-specific markup language. This transformation enables the subsequent Compiler module to generate high-quality visual outputs.

D. Compiler Module

The Compiler module is the final stage of the D.I.A.G.R.A.M. pipeline. Its primary role is to take the structured representation of the diagram, expressed in a markup language (e.g., Mermaid.js), and generate a high-quality diagram in a visual format such as PNG.

III. Flowchart Diagram Recognition

A. Internal Representation

In order to pass data through the components of the pipeline in a unified way, it was necessary to decide upon an internal representation of the flowchart diagram. It consists of **elements** and **relations**. The latter are linked to at least one element. Every element has a *category* and possibly some *inner* and *outer text*. Every relation has a *category* too, either a *source* or *target element* or both, and possibly *text* related to it.

This internal representation is generated from the image provided by the user by the Extractor module, while it is parsed and converted to a markup language of choice by the Transducer.

B. Classifier Module

1) Preprocessing: Due to the fact that various kinds of diagrams' images could be submitted to the system, it was decided to streamline a unified preprocessing pipeline in order to reduce the differences that our classifier model had to handle. A few problems were analyzed and the final preprocessing pipeline for the classifier module is composed of the following operators, in order:

- **Gray Scaling**: In order to remove colors from images, which was deemed an useless information for classifying diagrams.
- **Otsu Thresholding**: To binarize the image and reducing the information to handle.
- **Median Filtering**: Some images in our dataset - and in the use-case - were taken on checkered notebooks. The application of a median filter, with kernel size equal to three, totally removed these.
- **Perspective Correction**: The diagrams' images, sometimes, didn't occupy the full photo frame. Finding four keypoints (if possible) and stretching the image helped to fully employ the frame.
- **Padding**: This was necessary in order to keep the same net for every image. Changes in resolution would have implied changes in the number of paramters of the MLP layers.

Figure 2 shows the step-by-step process of preparing an input image for the network.

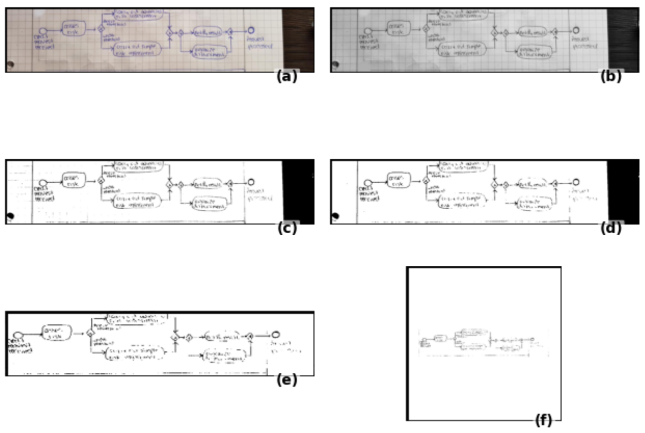


Fig. 2: The operations applied by the preprocessing module. **(a)**: The raw image, still colored. **(b)**: The image in gray scale. **(c)**: The binarized image, thanks to Otsu. **(d)**: The median filter result. Some previous thin lines are now no more. **(e)**: The image with a perspective correction. **(f)**: The image in a standard size, ready to enter the network.

2) Model: TODO

C. Extractor Module

1) Preprocessing: TODO

2) Bounding Box Detection: TODO

3) Content Recognition: After having determined all the bounding boxes of the diagram image, in order to correctly create the diagram internal representation, there's the need to link together text, elements and relations.

4) Arrow Orientation: To later correctly assign arrows to their source and target elements, the need to determine where the tip of an arrow lies. This task involves a few problems, among which determining exactly the arrow shape and its end - because sometimes other pen signs may be found in the bounding box of the arrow itself. [TODO: Dire l'approccio finale usato].

5) Text Digitization: Since the task of converting an image of text into the relative string has already been solved in various ways, we settled upon the Microsoft *Trocr Handwritten* Transformer model; more precisely, the *base* version of it. Every bounding box containing text that has been found is later passed to the model, and the resulting string generated by the model is linked to it.

6) Text Association: It can be assumed that every piece of text is linked to one and only one element or relation. So, every bounding box containing text is linked to the nearest element - it being either a relation or an element. The associations that require too long a distance are eliminated further in the algorithm, and the relevant text is discarded.

7) Element-Text Association: After that a text box has been assigned to an element, there's the need to understand if the text is either inside or outside the element and whether the text is actually referring to the given element or not. In order to do this, the overlap between the two bounding boxes is computed; if the overlap is over a certain threshold, heuristically decided, the association is kept. Otherwise, the distance between the two bounding boxes is computed - as the distance between their central point. If the distance is over a certain threshold, the association is kept; otherwise, it's discarded.

8) Relation-Text Association: After a text box is assigned to a relation, some computation is needed to fully understand how the text is associated to the relation itself. [TODO: Come trova source e target]. The relation bounding box is then split into three different sections. The overlap between the text bounding box and each of the three sections boxes is computed - then, the section with the highest overlap value is then assigned the text under consideration.

IV. Experiments

A. Classifier

The first architecture explored for the Classifier module was a Convolutional Neural Network (CNN) following AlexNet's architecture, but smaller in size. It had two Convolutional layers, each followed by a batch normalization and then by a max pooling layers. The first layer had only 4 filters, while the second one had 8. The output of the second layer was then flattened and passed through a fully connected layer which downsized the features to 128, followed by a ReLU activation function. Other two fully connected layer brought the number of output features down to the number of classes. Then, a softmax activation function was applied to the output layer, in order to obtain the probabilities of each class. The model was trained on our own dataset, with only three possible output classes: 'flowchart', 'graph' and 'other'; it was trained for 10 epochs, with an AdamW optimizer (with a learning rate of 1e-3) and a standard CrossEntropy Loss function.

We first tried such a small network because the features that had to be recognized were very simple, down to the basic shapes of the diagram elements. However, the results showed a non-decreasing value of the loss function of 0.9 - better than $-ln(3) \simeq 1.1$, which is random guessing, but still not satisfactory.

Judging this to be a problem in architecture size, we then tried adding another CNN layer with 16 filters; this one too followed by a batch normalization and a max pooling layer. The rest of the net was untouched; we also decreased the learning rate to 1e-4, in order to avoid overshooting the minimum of the loss function. The results, however, didn't show any kind of improvement, with the loss function still being around 0.9.

While analyzing the predictions, we came up with what was actually the problem: the dataset was heavily unbalanced, with the 'other' class being the most represented one, shadowing the other two classes. In order to solve this problem, we decided to use a weighted CrossEntropy Loss function, with the weights being inversely proportional to the number of samples in each class.

Algorithm 1 Cross-Entropy weight computation

```
Require: The classes' frequencies $f_1, f_2, ..., f_c$
1: $S \leftarrow \sum_{i=1}^{c} f_i$
2: $f_i \leftarrow (f_i/S)^{-1}$
3: $S \leftarrow \sum_{i=1}^{c} f_i$
4: $f_i \leftarrow f_i/S$
5: return $f_1, f_2, ..., f_c$
```

This final approach showed a significant improvement in the results, with the loss function decreasing to around 0.6 after 10 epochs of training, with a global accuracy score of 97% in the validation set.

B. Object Detection

TODO

C. Arrow Orientation

As we've seen in the previous section, the need to find the exact position of an arrow's head arises (see **Appendix A** for a full overview of the first approach).

TODO: Current approach.

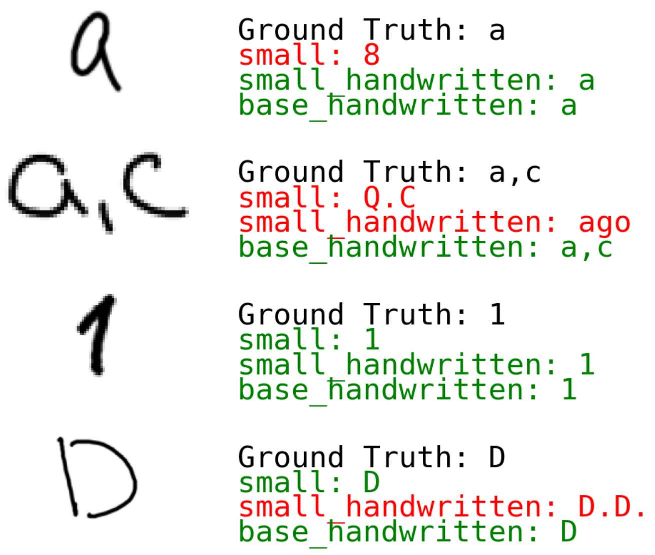
D. Text Digitization

In order to choose an appropriate model, some tests were done over our specific dataset employing some different, pre-made models. The average results over the dataset are shown in Table I.

TABLE I: Text Distance Metrics

Model	Hamming	Cosine	Euclidean
microsoft-trocr-small-printed	1.392	0.150	7133
microsoft-trocr-small-handwritten	**1.346**	0.100	3947
microsoft-trocr-base-handwritten	1.549	**0.057**	**3003**

Some examples of the different models applied to our flowchart diagram dataset can be seen in Figure 3 and in Figure 4.



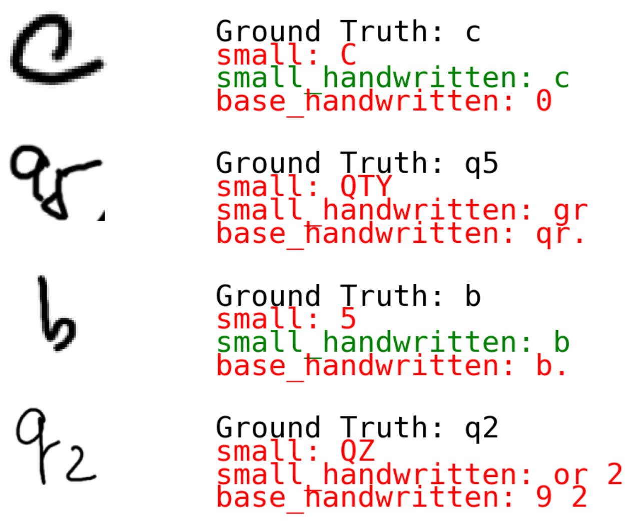
```
Ground Truth: a
small: 8
small_handwritten: a
base_handwritten: a

Ground Truth: a,c
small: Q.C
small_handwritten: ago
base_handwritten: a,c

Ground Truth: 1
small: 1
small_handwritten: 1
base_handwritten: 1

Ground Truth: D
small: D
small_handwritten: D.D.
base_handwritten: D
```

Fig. 3: Some random results of the text digitization over the proposed models.



```
Ground Truth: c
small: C
small_handwritten: c
base_handwritten: 0

Ground Truth: q5
small: QTY
small_handwritten: gr
base_handwritten: qr.

Ground Truth: b
small: 5
small_handwritten: b
base_handwritten: b.

Ground Truth: q2
small: QZ
small_handwritten: or 2
base_handwritten: 9 2
```

Fig. 4: Some results in which the *trocr-base-handwritten* fails.

V. Discussion

TODO

VI. Conclusion and Future Work

TODO

References

Appendix A Double Clustering

The first algorithmic approach proposed to solve this task - called *Double Clustering* - was as follows:

1) Apply a clustering algorithm to the bounding box.
2) Suppose that the biggest cluster belongs to the arrow; the other clusters are thought to contain useless elements for this task, and can be discarded.
3) Find the keypoints of the arrow using SIFT.
4) Find the two keypoints inside the arrow with maximum distance. Assume that these two points are closely related to the two ends of the arrow.
5) Run a template matching procedure inside the two clusters to find the head of the arrow.

For Step (1), we tried to employ both Spectral Clustering (see Figure 5) and DBSCan (see Figure 6).

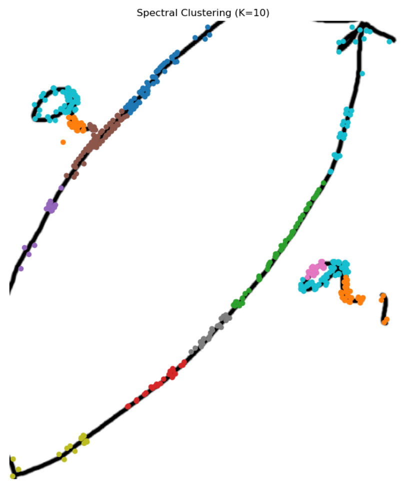


Fig. 5: The result of Spectral Clustering to one instance of an arrow bounding box in our dataset.

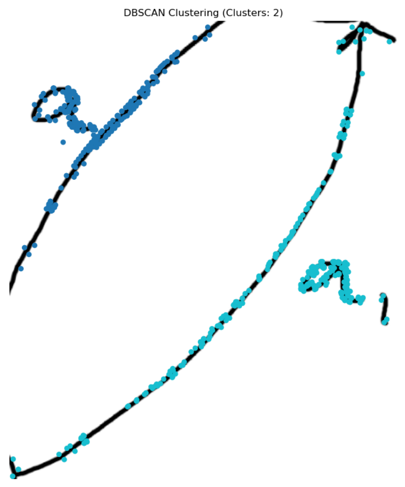


Fig. 6: The result of DBScan to one instance of an arrow bounding box in our dataset.

For Step (3), we used SIFT in order to find keypoints and some descriptors for them - necessary for the later template matching. During Step (4), we were able to correctly deduce the overall direction of the arrow (see Figure 7).

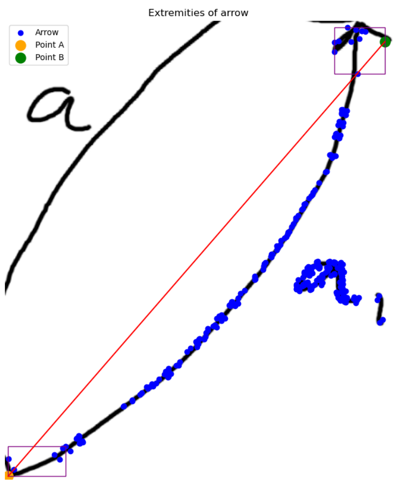


Fig. 7: The estimated direction of the arrow, finding the points that maximize the distance inside the cluster.

For Step (5), we had previously hand-picked some templates (see Figure 8).

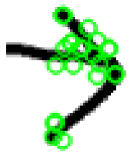


Fig. 8: An example of hand-picked template with SIFT descriptors.

However, this approach didn't work as expected (see Figure 9). [TODO: Spiegare perchè].

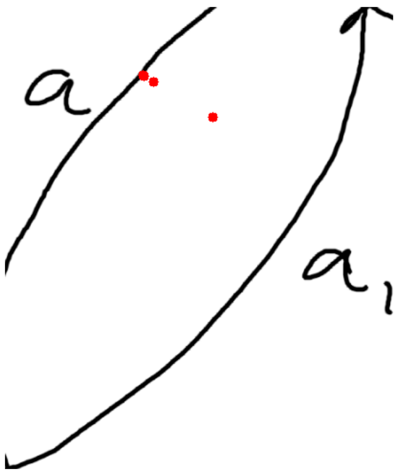


Fig. 9: The overall result of the algorithm.