Hand-Drawn Graph Recognition
ECE-549 / CS-543
Project Report

Debopam Sanyal (dsanyal2) Neeraj Gangwar (gangwar2) Bryan Huang (bryanh2)

1. Introduction

The goal of this project is to replicate the work done in Daly [1]. As described in the paper, our goal is to produce a system that can reproduce graphs from hand-drawn sketches. Daly does not have a public implementation, so we are reproducing the implementation ourselves from scratch. The Github repository contains iPython notebooks for each subtask that can be used to reproduce the results presented in this report.

The recognition process has three main steps - segmentation, classification, and domain interpretation. We have described these steps in their respective sections. Segmentation is discussed in section 2, classification in section 3, and the domain interpretation in section 4.

We consider two approaches to this problem. The method described by Daly, where recognition happens as the user draws, is the online approach. As an extension to the original goal, we also attempted to implement an offline approach where the entire graph is processed at once. We could not find any preexisting work relating to this topic for the offline approach as a whole, although there are some preexisting works for individual steps that we describe in their sections.

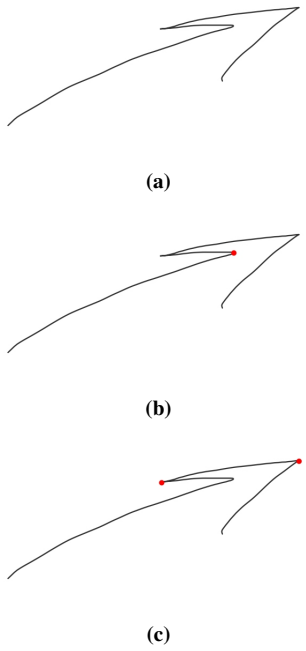


Figure 1: (a) a line and arrow head drawn in one stroke. (b) true segmentation point. (c) additional points with similar properties as true segmentation points.

2. Segmentation

The first step towards graph recognition is segmentation. This step identifies the segmentation points in the input graph. These points are fed into the recognition task to identify a component. Our implementation is based on the idea presented by Daly [1] and assumes that the entire component is drawn in one stroke. However, drawing multiple components in the same stroke is acceptable for our implementation.

The segmentation points identify the transition from one component to the next. These are referred to as true segmentation points. However, there are certain challenges in the segmentation task. These challenges arise from the fact that there might be additional points in a component with similar geometric properties as true segmentation points. This is shown in Figure 1. The challenges can be divided into two buckets: false-positives and false-negatives. The points that are incorrectly identified as segmentation points contribute to false-positives. These cases are less severe as they can be handled in the classification step. The points that are true segmentation points but are missed by the segmentation algorithm are false-negatives. These cases are more severe as we cannot recover from these errors. The goal of the segmentation algorithm would be to minimize false-positives while ensuring that there are no false-negatives.

2.1. Algorithm

We explored two segmentation algorithms for the online recognition problem - curvature-based and speed-based.

2.1.1 Curvature-Based Segmentation

The curvature-based segmentation algorithm is based on the observation that there are abrupt changes in curvature at segmentation points [1]. This is based on the properties of transition that can occur in the three types of graph components. This is shown in figure 2.

Before the curvature calculation, some pre-processing of input data is required. We use iPyCanvas [2] to capture the coordinates of the drawn components. The number of coordinates captured depends on the drawing speed. If the drawing speed is higher in certain regions, the number of points captured will be higher in those region. This count is normalized by considering the distance between two points. If the Euclidean distance between two consecutive points, P_i and P_{i+1}, is less than a threshold, d_t, we drop P_{i+1}. Here, d_t is a hyperparameter that needs to be trained based on the input data.

The *curvature* of a point P_i is computed by calculating the angle between the vectors formed by joining P_{i-s} and P_i and P_i and P_{i+s}. s is the second hyperparameter.

$$C_s(P_i) = \arccos\left(\frac{\overrightarrow{P_{i-s}P_i} \cdot \overrightarrow{P_iP_{i+s}}}{\|\overrightarrow{P_{i-s}P_i}\| \, \|\overrightarrow{P_iP_{i+s}}\|}\right) \tag{1}$$

Once the curvature is computed for all captured points, we compute *abnormality*. This metric is used in identifying the abrupt and significant changes in the curvature. The abnormality of a point P_i is defined by how the curvature at that point differs from the average curvature of the surrounding points.

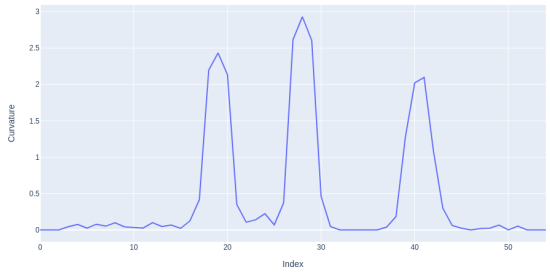
$$A_{s,w}(P_i) = C_s(P_i) - \frac{\sum_{j=a}^{i-1} C_s(P_j) + \sum_{j=i+1}^{b} C_s(P_j)}{b-a} \tag{2}$$

where $a = max(1, i-w)$ and $b = min(n, i+w)$.

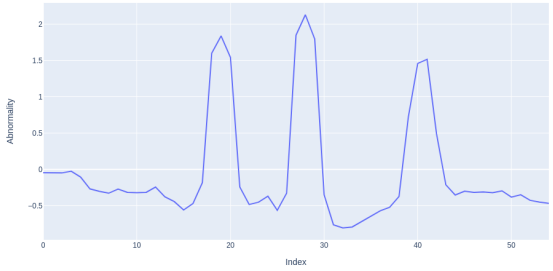
w is the third hyperparameter. Next, we need to find ranges of points where $kA_{s,w}(P_i) > 1$ and select the point with maximum abnormality in each range as the segmentation point. The multiplication factor k is the fourth hyperparameter.

2.1.2 Speed-Based Segmentation

This algorithm is based on the observation that the drawing speed reduces while drawing corners. We implemented and experimented with a simplified version of the algorithm presented by Wang et al. [3].



(a) Curvature



(b) Abnormality

Figure 2: Curvature and abnormality plots for the components shown in figure 3a. The peaks represent the segmentation points.

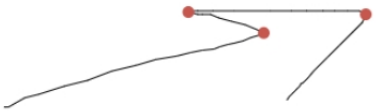
For every coordinate captured by iPyCanvas, we also recorded a timestamp. The *speed* of stylus for a point P_i is given by

$$v(P_i) = \begin{cases} \frac{\|\overrightarrow{P_{i-i}P_i}\|}{t_i - t_{i-1}} & 0 < i < n-1 \\ 0 & i = 0; i = n-1 \end{cases} \tag{3}$$

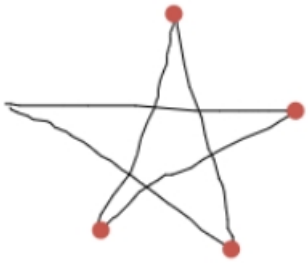
Once the speed is computed for all captured points, we find ranges of points where $v(P_i) < V_t$ and select the point with minimum speed in each range as the segmentation point. V_t is a hyperparameter here.

After computing the potential segmentation points, we perform a post-processing step to drop the points that are close to each other. This step is similar to the pre-processing step discussed in the curvature-based segmentation algorithm and has a hyperparameter d_t.

As we would see in subsection 2.2, this method is not super accurate and produces quite a few false-positives. But it is very fast compared to the curvature-based segmentation. In general, the component classification step can take care of false-positives. So, if a usecase requires faster computation,



(a)



(b)

Figure 3: Curvature-based segmentation algorithm. The segmentation points are highlighted in red.

this method can be used with a good component classifier to reduce the overall computation time.

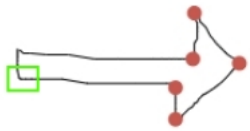
2.2. Experiments

To test the segmentation algorithms, we drew some components and shapes manually on iPyCanvas and validated the results.

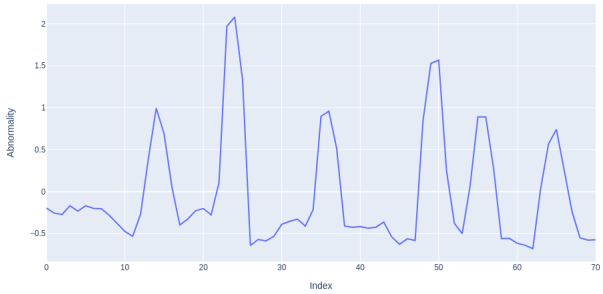
Figure 3 shows the detected segmentation points for the curvature-based segmentation algorithm. Figure 2 shows the curvature and abnormality plots for the stoke in figure 3a. We also observed cases where a segmentation point was missed by this algorithm. This is shown in figure 4. The abnormality plot shows all peaks correctly, but the last peak is below the threshold. Hence, the hyperparameter tuning is a critical step for this algorithm. Based on our experiments, the following values of hyperparameters worked best - $d_t = 2$, $s = 3$, $w = 12$, and $k = 1.2$.

Figure 5 shows the detected segmentation points for the speed-based segmentation algorithm. Figure 6 show the corresponding speed plots. It is apparent that this algorithms produces a lot of false-positives. Based on our experiments, the following values of hyperparameters worked best - $V_t = 100$ and $d_t = 10$

The execution time for both algorithms is shown in table 1.



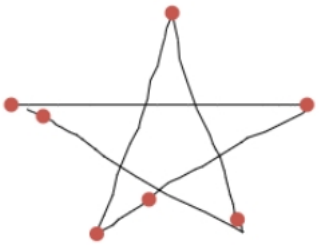
(a)



(b)

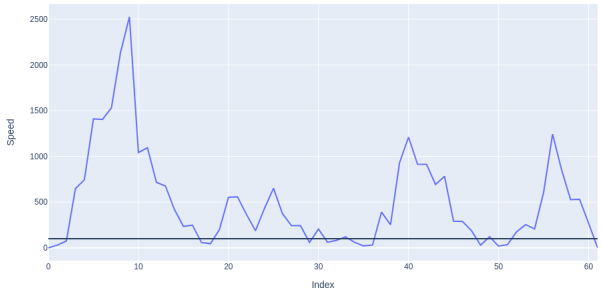
Figure 4: False-negative with the curvature-based segmentation: (a) The green box shows a false-negative. (b) The abnormality graph for the stoke.

(a)

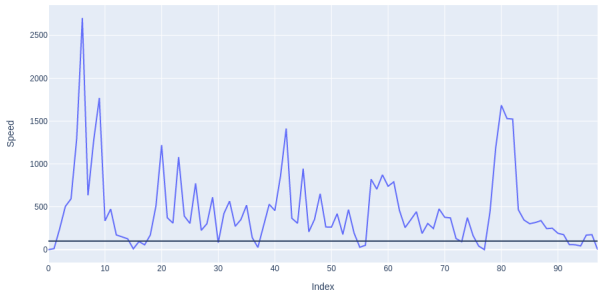


(b)

Figure 5: Speed-based segmentation algorithm. The segmentation points are highlighted in red.



(a)



(b)

Figure 6: Speed plots for (a) arrow 5a (b) pentagram 5b. The black line shows the threshold. Points below this line are potential segmentation points.

Stroke	Curvature-Based	Speed-Based
Arrow	3.38 ms ± 28.9 µs	555 µs ± 15 µs
Pentagram	8.77 ms ± 779 µs	973 µs ± 24.4 µs

Table 1: Execution time

Based on the number of false-positives and false-negatives, we used the curvature-based segmentation algorithm in our graph recognition pipeline.

3. Classification

After segmentation is completed, we run classification on each segment. This is to determine the probability that a given segment is one of four types - vertex, self-loop, edge, or arrow. These probabilities are used in domain interpretation step to reconstruct the original graph.

3.1. Original Algorithm

The first version of classification we implemented was a recreation of the original paper. In order to determine the segment's probability distribution, we find five parameters. These parameters were chosen by Daly [1], based on the alpha shape and convex hull of the segment.

To compute the alpha shape and convex hull, we use the Alpha Shape Toolbox library [4], as well as SciPy's ConvexHull implementation [5]. These libraries provide tools to easily find the area, perimeter, and points of these shapes.

3.1.1 Circumscribed circle

The first parameter is used to distinguish vertices and loops from other segments. This parameter x_1 is computed as

$$x_1 = \frac{\text{Area of Convex Hull}}{\text{Area of circumscribed circle of Convex Hull}} \tag{4}$$

The convex hull is not necessarily cyclic, so we define the circumscribed circle as a circle with diameter equal to the distance between the two farthest points in the convex hull.

3.1.2 Inscribed triangle

The second parameter is used to distinguish arrows. This parameter x_2 is computed as

$$x_2 = \frac{\text{Area of largest inner triangle with angles over } 20°}{\text{Area of Convex Hull}} \tag{5}$$

We use the method by Dobkin and Snyder [6] to find the area of the largest inner triangle.

3.1.3 Alpha shape ratio

The third parameter is used to distinguish line segments. This parameter x_3 is computed as

$$x_3 = \frac{\text{Perimeter of alpha shape with } \alpha = 25}{500 \cdot \text{Area of alpha shape with } \alpha = 25} \tag{6}$$

3.1.4 Perimeter

The fourth parameter is used to distinguish vertices and arrows from other segments. This parameter x_4 is computed as

$$x_4 = \frac{\text{Perimeter of Convex Hull}}{500} \tag{7}$$

3.1.5 Disjoint shapes

The final parameter disqualifies any strokes that are not properly segmented. This parameter x_5 is computed as

$$x_5 = \text{Number of disjoint regions in alpha shape over 50 pixels apart} - 1 \tag{8}$$

The weights applied to each parameter reflect traits held by classes of stroke. For example, vertices and self-loops are almost circular, so they assign high positive weight to the circumscribed circle parameter. For testing purposes, we use the original weights trained by Daly [1]. Notably, the weight for parameter x_5 is not trained, and instead set to always be -1000, to minimize the probability distribution of any shape with multiple disjoint regions.

3.2. Offline Implementation

Classification, unlike segmentation and domain interpretation, can be accomplished without needing the chronological order of drawn points. As long as the collection of points is present, features can be extracted without any significant difference in effectiveness. Within our implementation, the only feature computed with exponential complexity is the circumscribed circle, and that feature can be computed in polynomial time through rotating calipers.

There is the consideration of the input points being too granular. The original paper handles this by removing any line segments greater than a given distance. This implementation requires chronological order of points, but we can accomplish a similar level of granularity through downsampling the input pixels.

3.3. CNN Approach

While classification does not need significant improvements to be effective both online and offline, we explored training a CNN as another method for classification. Daly's implementation of classification was trained on manually chosen features, so we attempted to see how features extracted by CNN would complete the same problem.

Creating and labelling data would take a significant amount of time for an exploration, and there are no public data sets for this particular problem that we could find. Because the classes of graph components are so simple in shape, we can use a data set like MNIST, with only a few changes. MNIST is a very basic and commonly used data set, so almost any network can easily achieve a very high accuracy. However, high accuracy on the MNIST test set does not translate to high accuracy with our task.

This method cannot differentiate self-loops and vertices as effectively as the original paper. For this reason we change number of classes to 3 rather than 4, combining vertices and self-loops into one class. The primary difference between the two classes is the perimeter of the stroke. The original implementation is able to differentiate by using perimeter as a feature. We would need to make the network scale-aware, and further preprocess the data set.

Another issue with this method is that domain interpretation requires additional information not used in classification that the CNN cannot extract. The hand-picked features for classification are similar enough to the features needed for domain interpretation that it is efficient to compute simultaneously, which cannot be done by the CNN.

Line thickness also negatively affects this method. In the online case, the original method does not need image data and extracts points through the canvas. In the offline case, the original method can use erosion to reduce a stroke to a list of relevant points. However, in the CNN method, the MNIST dataset contains training images with particular line widths. Assuming that a drawn graph has consistent line width throughout, after resizing strokes with different sizes, the resulting CNN inputs would have different line widths. In our demonstration notebook, we use a fixed canvas size and line width, but in experiments, we found that too small of a line width would make classification incorrect.

Due to all of the above considerations, we found that stroke classification is better suited to using hand-picked geometric features than recognition through CNN.

3.3.1 Dataset

We used a modified version of the MNIST dataset. All entries labelled with a digit other than 0, 1, and 7 were removed from the dataset. The resulting dataset contained 18,930 training images and 3,134 test images. We randomly rotate the training input within a range of 360 degrees, in order to train for all possible orientations.

3.4. Experiments

Similarly to the segmentation algorithm, we tested classification by drawing single strokes on the iPyCanvas. One example is shown in Figure 7.
For the CNN method, we used the test dataset described above.

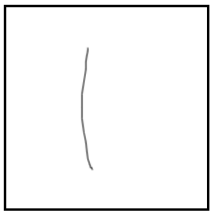


Figure 7: Example segment. $p_{vertex} = 0.0000076, p_{arrow} = 0.256, p_{edge} = 0.996, p_{loop} = 0.000086$

4. Domain Interpretation

After we compute the probabilities from the classification section, we need to produce a final interpretation of the graph in terms of edges, vertices, arrows, self-loops, connections between edges and vertices and connections between arrows and edges. To do this we cannot use the trivial method of partitioning the graph into segments (from our segmentation step) and simply solving for the partition that produces the largest sum of classification probabilities (from the classification step). This is mainly because there is significant interaction between components of a graph. The design that we follow rests on the idea that the connections between the components influence the selection of the best interpretation.

4.1. Scoring Subgraphs

We found it useful to rank candidate graphs by scores that are formed from a range of metrics like sum of classifier probabilities $g(p)$, total number of components $C(p)$, number of connections $D(p)$ and number of missing connections $M(p)$. Thus, the combined score is simply:

$$V_{A,B,C}(p) = g(p) + A.C(p) + B.D(p) + C.M(p) \quad (9)$$

where $g(p) = \sum_{(a_i,b_i,c_i)\in p} f(a_i, b_i, t_i)$
The function f is determined by the classification probabilities. The score however is 0 when any component in p has a probability less than 0.2. To properly assess the connections, we had to establish a set of machine specifications (detailed in src/interpretation.py of our code) for our components where we could find the distances, say, between an edge and vertex or an edge an arrow.

The recognition step provided some unique challenges. Here, an exhaustive search for candidate graphs had to be done because in the classification step we returned probabilities of all possible components. Since there are four different components that we consider: arrow, edge, vertex, loop, any segment can be one of the four components. Hence, the total number of candidate graphs to be considered at any step in the online setting is $4^{n_{segments}}$. We obtain $n_{segments}$ from the segmentation step where segmentation points are calculated from strokes. For each segment, once classification is done, we first create a Cartesian product of components to include all possible combinations. Then, each combination is converted to conform with our machine specifications. Next, each combination is added to the locked-in graph to form the subgraph at step i. The score function assesses each such subgraph and assigns a final score. This final score is calculated using the connections formed between components in the classified graph and using the sum of probabilities acquired from the classification step. We discard the sum of probabilities for any subgraph if any of its components had less than 0.2 probability. This essentially means we filter out the subgraphs with extremely unlikely components. The candidate subgraphs are sorted in descending order of their final scores. The highest scoring subgraph is treated as the interpreted graph for that step. Subsequently, the interpreted graph becomes the locked-in graph for the next step of handling user strokes and the pipeline is run all over again.

In addition to the classification probability, the scoring depends on the connections too. Since the number of components tend to be over-counted by this method, it has a negative coefficient A. The number of connections made has a positive coefficient, B, because ideally, we want our interpretation module to detect as many connections as possible. A slightly more complex metric is the number of missing connections. To calculate it, we must penalize the graphs for having edge points not connected to vertices, arrows not connected to edges and self-loops with different starting and end vertices. The penalty is ensured again by a negative coefficient C. Finding loops connected to different vertices was challenging as looking for the same loop in two different connections was not an easy task. We resolved this by doing an element-wise comparison of the loop points.

4.2. Connections

A vertex-edge connection is tested for every vertex and edge pair. We can use the center and radius of the vertex to find the distance from the endpoints of the edge as edges are given as a set of target points that represent a smooth line. Computing the Euclidean distance between the center and each of the two endpoints of the edges gives us an idea of whether the connection is plausible. Of course, we must subtract the radius of the circle itself and threshold the resulting distance to discard the large distance values. We maintain a score q for each edge endpoint that is initialized to infinity and updated if and only if a new vertex is found

for the end point such that it's distance falls below the threshold and it is less than the previous q (negative values are clipped to 0).

The arrow-edge connection algorithm is more complex because it involves triangles (machine specification for arrows) and lines (edges) that must intersect one of the sides of the triangle. Again, connection is tested for every arrow and edge pair (from the endpoints of the edge). Here, we first extend the edge sequence by adding a target point before the end point such that smoothness is still maintained. The distance between the new and the old point is γ . Now, we test for the segment between each consecutive pair of the extended edge. If the segment intersects with a side of the triangle, then a condition must be satisfied: the distance between the endpoint and the point of intersection must be under the threshold or the distance between the endpoint and the point opposite to the intersection side of the triangle must be under the threshold.

The central portion of domain interpretation is calculating the vertex-edge and arrow-edge connections. The vertex-edge calculation was relatively simple since the center is provided by the classification module. The next steps involved comparing the distances between the edge endpoints and the center of the circle. Based on a threshold, it is determined whether the edge endpoint is close enough to the vertex to form a connection. Since an endpoint can be connected to at most one vertex, we iteratively update the connection and finally end up pairing the edge with the vertex that has the minimum distance to the circumference of the vertex (or circle). Constructing an algorithm for arrow-edge connections was much more difficult. Since an arrow is represented with a triangle, choosing the correct orientation of the triangle is very important. Finding the intersection point of the edge and a side of the triangle helps in getting the correct orientation. Adding an extra point after the original endpoint made sense as it brought the edge closer to the correct corner of the arrow. This made it easier to do a threshold-based check on the distance between the new endpoint and the correct corner or the new endpoint and the midpoint of the intersecting side to determine the validity of the connection. Again we have the constraint of an arrow being able to connect to at most one edge endpoint. The new endpoint was added utilizing the nearby slope of the edge as follows:

$$e_0 = (e_{1x} \pm \frac{\gamma}{\sqrt{m^2+1}}, e_{1y} \pm \frac{m\gamma}{\sqrt{m^2+1}}) \qquad (10)$$

4.3. Cost Calculation

We followed algorithm 1 in [1] to calculate the cost of a hand-drawn graph's interpretation using our algorithm. The cost is nothing but the number of subgraphs the algorithm gets wrong. This means that the lower the cost, the better the performance. When we say subgraph, we mean the subgraph obtained after adding the strokes inputted in step i to the subgraph that was locked-in at step $i-1$. The concept of a locked-in graph is important in an online setting as this means that the subgraph that has already been interpreted cannot be changed. This follows from the online goal of analyzing the graph at most five components at a time. A fundamental struggle in this step was to come up with a good function to check whether the interpreted subgraph is isomorphic to the intended subgraph as required by the algorithm. Since it is not tractable to try all permutations of vertices, we opted to focus on a few well-known heuristics. Checking for the number of vertices and edges and the vertex degree sequences of both the graphs often works well for simple graphs. We used just these three conditions to detect isomorphisms. Other heuristics like checking whether the degrees of adjacent vertices match were complex and hence we did not try them. We also checked isomorphisms only for the undirected versions of directed graphs as it is very unlikely in our case that directed graphs are isomorphic, but their undirected versions are not.

In every step, we select the most promising candidate graph according to their scores. However, if an isomorphism exists, we select the isomorphism as the representation of the interpreted subgraph. The cost is 0 if the isomorphism was the best candidate graph, otherwise it is 1. If there exists no isomorphisms, then we penalize all the remaining pairs of new strokes and subgraphs.

4.4. Experiments and Results

Graph	Avg. Cost	Steps
3	2.4	3
4	2.0	3
6	1.6	2
7	2.8	3
9	2.6	3
11	3.6	4
13	5.6	6

Table 2: Average cost of interpreting 7 graphs with 5 trials each

We conducted our experiments using the end-to-end pipeline (shown in the IPython notebooks in the repository) including segmentation, classification, and interpreta-

tion for each step, where a step signifies a set of user strokes that are analyzed by this pipeline and added to the locked-in graph to form a new locked-in graph. We selected graphs 3, 4, 6, 7, 9, 11 and 13 from the Mechanical Turk Experiment 1 as listed in the [1] for our tests. Each graph was drawn five times with five components added in each step, except the last step where five or less components were added. The five components added in a particular step of a graph were not the same across trials; for example, we didn't always add the same five components for step 1 of graph 7. The interpreted graph after all the steps was a list of vertices, edges, arrows and loops appearing in the form dictated by our machine specifications. Table 2 shows the average cost of interpreting each graph. Naturally, the average cost is going to be more for graphs that require more steps to be drawn. Hence, the average total cost does not particularly help in comparing the algorithm's performance across graphs. Therefore, we constructed a plot (Figure 8) which shows the average cost of steps 1 and 2 for all the seven graphs. Since not incurring cost at step i requires the cost at step $i-1$ to be 0, the average cost for step 2 will always be more than or equal to the average cost of step 1 for a particular graph. We also trained within the parameter space stated in [1] to obtain the best coefficients of the terms used in the candidate subgraph scoring function. Several combinations yielded the lowest cost, but we used A = -1.1, B = 0.3 and C = -0.4.

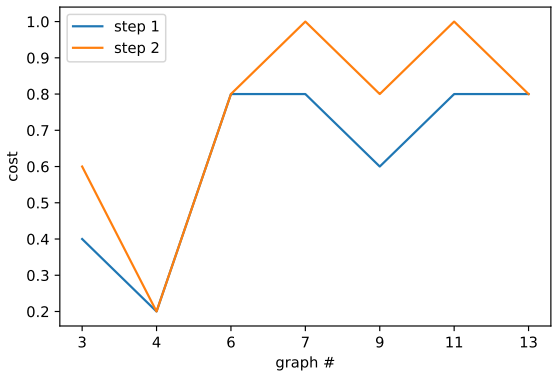


Figure 8: average cost of step 1 and step 2 for the 7 graphs

5. Analysis

Table 2 shows that our pipeline did not perform exceptionally well. This was mainly because the cost, as defined in [1], is a very harsh penalty. It is binary with no partial reward for some or most of the subgraph being correct. Since the correctness of an interpreted subgraph is determined by whether it is isomorphic to the intended subgraph, the cost at a particular step will 0 if and only if there is an isomorphism between the two. Other issues that affect our algorithm is the fact that we reuse the thresholds from [1]. These thresholds may not be appropriate for our drawing setting in iPyCanvas [2]. Training to find our own values of the coefficients in the scoring function was not as helpful as expected. A reason could be that the search space for the parameter grid search was not right. Another potential weakness in our method is probably the inconclusive isomorphism function. Figure 8 shows that subgraphs with vertex and edges are more easily interpreted than those with loops and arrows. This was expected as circles for vertices and smooth lines for edges are much easier to break into segments and classify than self-loops and triangles for arrows.

6. Conclusion and Future Work

In this project, we worked on the online graph recognition problem. For offline recognition, we tried to use the same pipeline by converting an offline graph to a sequence of coordinates. But this approach was not successful as the current pipeline heavily relies on the ordering of the coordinates. The offline recognition problem can be modeled as a learning problem and deep learning approaches can be used to solve it.

The segmentation algorithms discussed in this report heavily rely of hyperparameter tuning. The values of these parameters can vary from person-to-person. Rather than finding the parameter values that work for everyone, we can tune these parameter for an individual by taking feedback from the user. The speed-based segmentation can be explored further. To reduce the false-positives, an abnormality computation step similar to the curvature-based segmentation can be added to overcome the noise.

For classification, we found that it was a task well-suited to multiple pipeline configurations, only needing to adjust the input format. The geometric feature approach described in the original paper was the most effective. The problem is simple enough that attempting convolutional methods introduces significant complexity for minimal increase in efficacy.

Classification is simple enough not to need an approach more complex than the features discussed here, but it is closely related to segmentation. Future work could include an object detection method to perform both tasks simultaneously.

Interpretation depends heavily on segmentation and classification for accuracy. The main portions of interpretation are establishing vertex-edge and arrow-edge connections, scoring candidate graphs, and calculating the cost of an interpreted graph. From our results, we can see that vertices

and edges were easier to deal with than arrows and loops. Future work would involve improving on the isomorphism function that we have. Another potential area is to experimentally determine threshold values. Another obvious area of development is building the offline method. There are some fundamentally different aspects to the offline method like the recognizer will only act on the graph after completion of all strokes and intermediate subgraphs do not have to be maintained in a locked-in graph. We found out that the cost function is not the best metric to evaluate our methods. It assigns full cost even to a graph with just one incorrect component. Thus, developing another quantitative performance evaluation metric for hand-drawn graphs is something that we want to work on in the future.

In the current pipeline, the output needs to be evaluated by inspecting probabilities and costs. It can be replaced with a user interface to visualize the output. It is a required feature for the end-user and will significantly reduce the time to analyze the performance of this pipeline.

7. Member Roles

Neeraj: segmentation, Bryan: classification, Debopam: interpretation and training. All of us worked on integrating the subparts into a whole. We used Github to maintain and share data (Github Repository). We met once a week to discuss updates and issues.

References

[1] Katharine M Daly. *Hand-drawn graph problems in online education*. 2015.

[2] Martin Renou. *ipycanvas - Interactive Canvas in Jupyter*. 2021 [Online]. URL: https://github.com/martinRenou/ipycanvas.

[3] Guanfeng Wang et al. "Segmentation of Online Freehand Sketching Based on Speed Feature". In: *Mathematical Problems in Engineering* 2021 (2021).

[4] K. Bellock. *Alpha Shape Toolbox*. 2021 [Online]. URL: https://alphashape.readthedocs.io/en/latest/.

[5] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. 2001–. URL: http://www.scipy.org/.

[6] David P. Dobkin and Lawrence Snyder. "On a general method for maximizing and minimizing among certain geometric problems". In: *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*. 1979, pp. 9–17. DOI: 10.1109/SFCS.1979.28.