Image Recognition Research Report - Team Synergetics

IFB399 – Capstone Project

Gregory Mandall - n10757163

Alex Achille – n11100842

Yen Do – n10616748

Jack Girard – n11252511

Submitted: October 2023

TABLE OF CONTENTS

1. INTRODUCTION 2

2. PHASE 1 - OPENCV & JAVASCRIPT 2

2.1 FUNCTIONALITY ACHIEVED 3

2.2 ROADBLOCKS 7

2.3 RECOMMENDATIONS 8

3. PHASE 2 - OPENCV & PYTHON 9

3.1 FUNCTIONALITY ACHIEVED 9

3.2 ROADBLOCKS 11

3.3 RECOMMENDATIONS 11

4. PHASE 3 - TENSORFLOW & PYTHON 12

4.1 FUNCTIONALITY ACHIEVED 12

4.2 ROADBLOCKS 18

4.3 RECOMMENDATIONS 19

5. FUTURE RECOMMENDATIONS AND CONSIDERATIONS 21

1. INTRODUCTION

A research report has been generated to comprehensively document all avenues investigated for the implementation of image recognition software aimed at the successful identification of digital graph-based diagrams. This report will illustrate the functionality achieved, the roadblocks, and possible recommendations that should be considered to achieve the successful implementation of image recognition into S23M's model editor. This report will include three distinct phases covering the initial image recognition development using OpenCV in JavaScript, the development phase using OpenCV in Python, and the shift to the current development stage in TensorFlow using Python.

2. PHASE 1 - OPENCV & JAVASCRIPT

The primary goal in Phase 1 was to integrate an image recognition module into an existing graph editor. After careful evaluation of available technologies, the team opted for OpenCV.js due to its existing shape recognition library and its compatibility with JavaScript which aligns seamlessly with the client's ReactJS-based graph editor. The team's achievements encompassed setting up a test environment, applying basic image recognition techniques for shape detection, and exploring the potential of OpenCV.js.

However, the team encountered several challenges during this phase. Contour coordination issues impeded precise coordinate extraction from detected contours. Additionally, the approach to vertex detection inadvertently hindered the accurate detection of arrows and textual labels which is crucial for comprehensive graph representation. Moreover, limited documentation and resources available for development in OpenCV.js posed hurdles in extracting information from the detected objects. To address these challenges, the team derived key recommendations for the subsequent phases. These include leveraging the success of the prototype, adapting the approach to handle complex graph models, exploring OpenCV with Python for its extensive documentation and resources, and finding innovative solutions for information extraction challenges. The insights gained in Phase 1 have formed the foundations for a more informed and focused approach in subsequent phases, aiming to refine the image recognition module and deliver a powerful, accurate, and versatile tool for graph analysis.

2.1 FUNCTIONALITY ACHIEVED

Phase 1 accomplished significant milestones in the pursuit of seamlessly integrating image recognition capabilities into the existing graph editor. The primary objectives were to establish a test environment, implement basic image recognition techniques, and showcase the potential of OpenCV.js and JavaScript within this context.

Test Environment Setup:

A pivotal accomplishment was the creation of a dedicated JavaScript website that served as a testing ground for image recognition functionalities, shown in Figure 1. This interactive platform allowed users to select images and manipulate detection parameters, offering a tangible glimpse into the power and potential of integrating image recognition within the graph editor. To ease debugging and verification, the team utilised OpenCV's imshow() function to display the final image with bounding boxes, providing visual cues for successful shape detection. The choice of JavaScript facilitated a smooth integration into the client's ReactJS-based environment, aligning with their technological preferences.

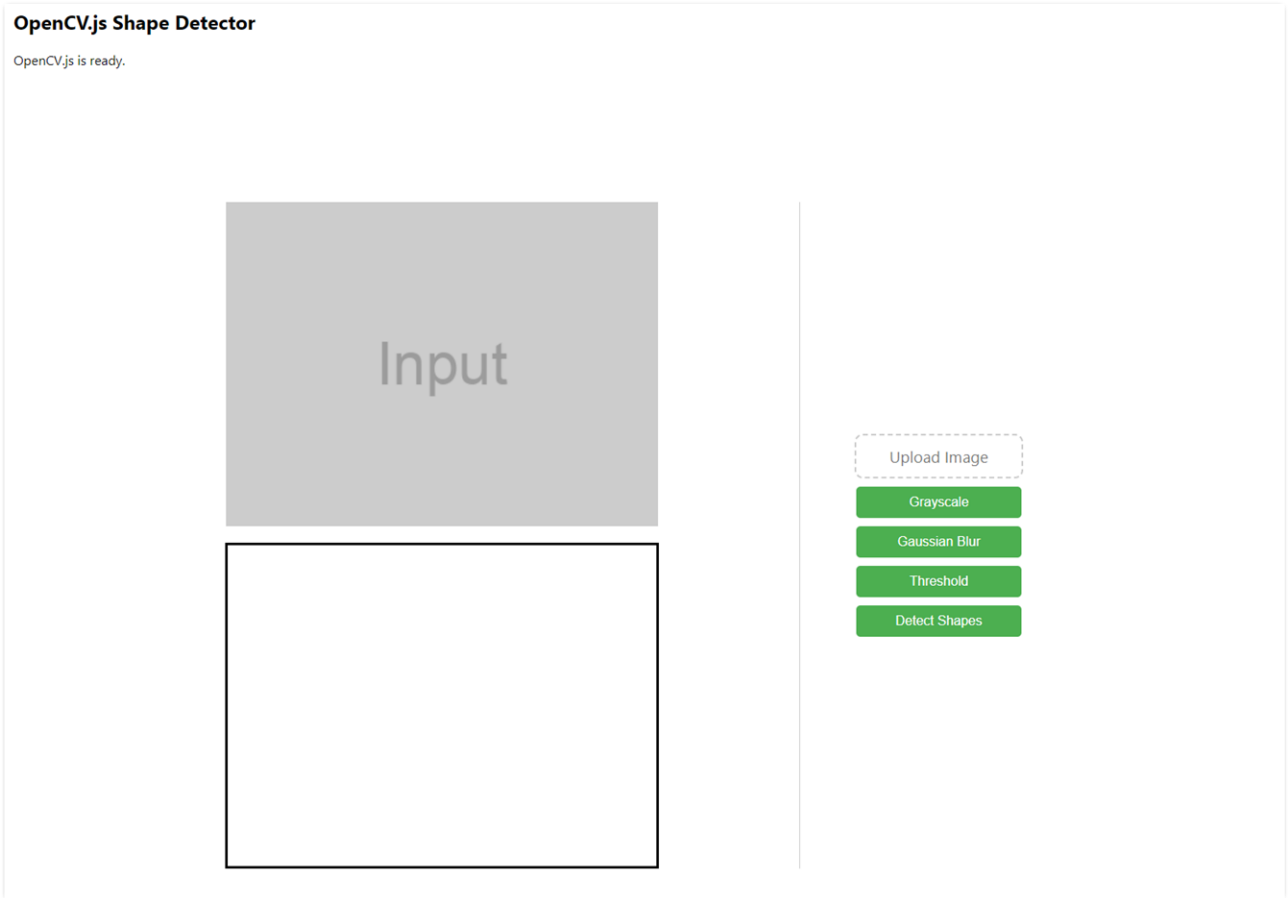

Figure 1: Prototype Shape Detection Application.

Basic Image Recognition Techniques:

Leveraging OpenCV.js, the team applied fundamental image recognition techniques to detect and analyze shapes and contours within graph images. These techniques included:

- **Image Preprocessing:** OpenCV's imread() function loads images, initiating the preprocessing pipeline. Preparing images through preprocessing techniques enhanced the quality of inputs for subsequent processing steps.
- **Grayscale Conversion:** To simplify subsequent analysis, the team employed OpenCV's cvtColor() function to convert images to grayscale, reducing the dimensionality of the data while retaining essential information. Figure 2 shows the results of the grayscale conversion operation being applied to a sample image in the prototype.

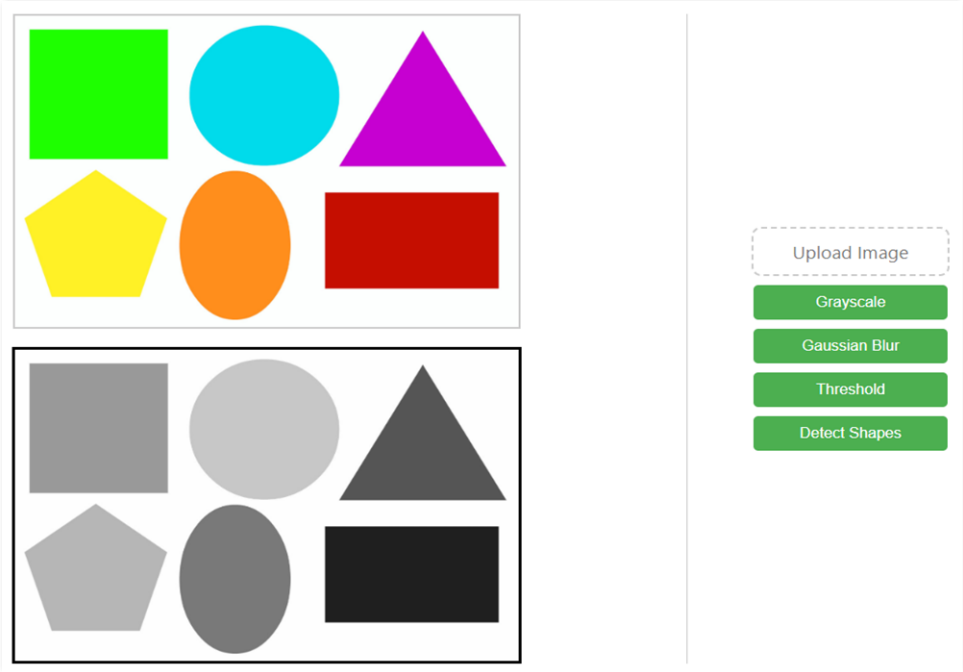

Figure 2: Outcome of Grayscale Operation Applied to a Sample Input Image.

- **Thresholding:** Employing OpenCV's threshold() function, the team transformed grayscale images into binary representations. This effectively isolated features of interest based on pixel intensity, as shown in Figure 3. The team experimented with various thresholding methods such as ‘binary’, ‘binary inverted’, ‘trunc’, ‘to-zero’, and ‘to-zero inverted’. The team found that using parameters with values such as 200px and 255px, along with the ‘binary’ thresholding method, produced the most optimal results for shape detection purposes.

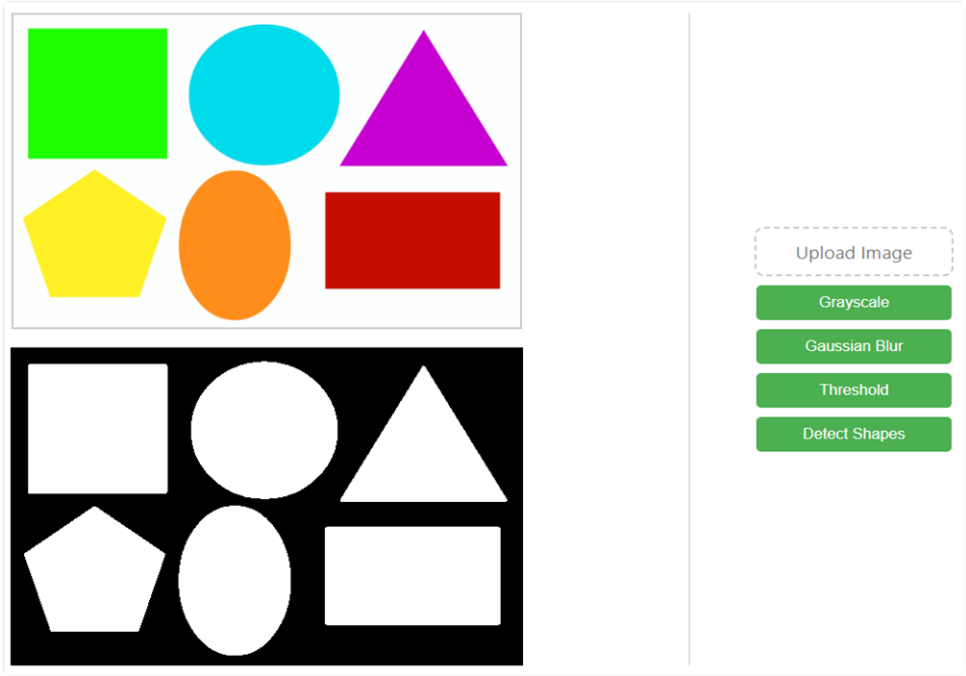

Figure 3: Outcome of Threshold Operation Applied to a Sample Input Image.

After processing the image, the team applied findContours() to start the detection process. These techniques collectively enabled the team to detect and differentiate various shapes, laying the foundation for more sophisticated analyses in the future. As shown in Figure 4, the prototype was able to effectively detect and label various shapes provided they are neatly spaced out in the image.

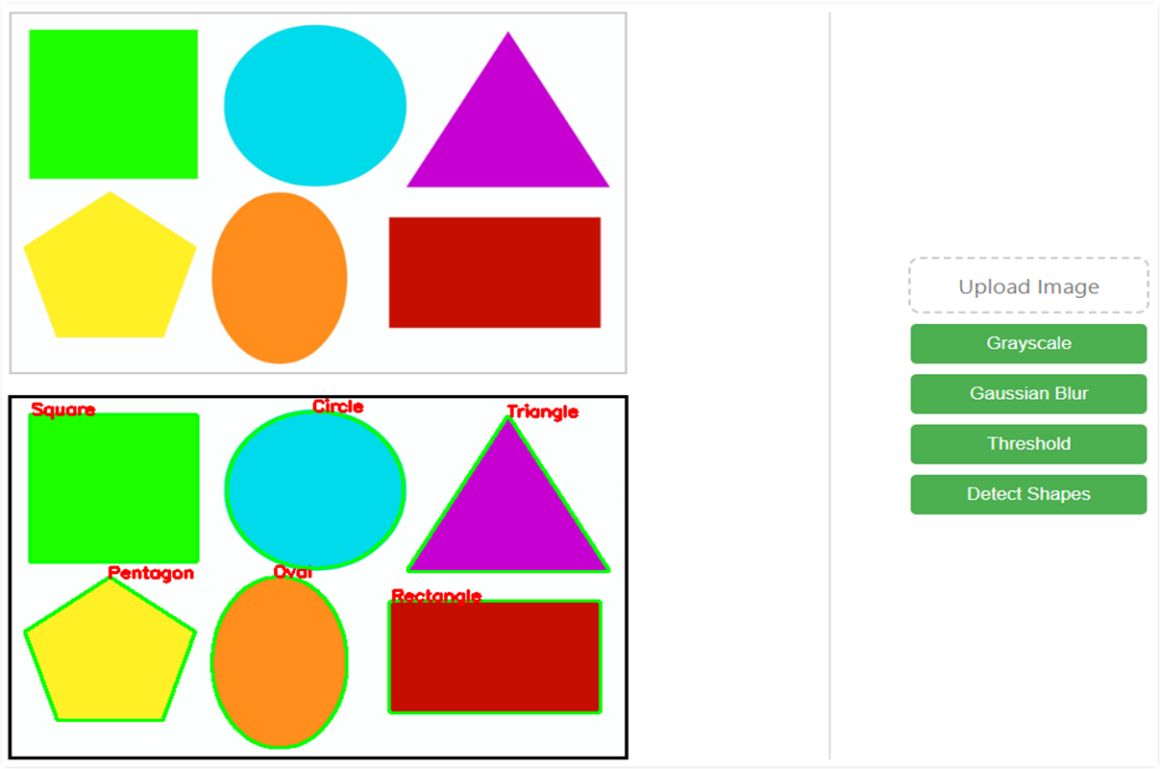

Figure 4: Shapes Detected in the Preprocessed Sample Input.

Advanced Shape Separation:

The team employed advanced detection techniques to further refine shape recognition capabilities. Adaptive Thresholding and Morphological Operations played a pivotal role in isolating complex components within the images.

- **Adaptive Thresholding:** `adaptiveThreshold()` with the parameters `ADAPTIVE_THRESH_GAUSSIAN_C` and `THRESH_BINARY_INV` were utilised in scenarios where a global threshold doesn't yield satisfactory results due to varying illumination conditions. This technique divides the image into smaller sections and calculates local thresholds, ultimately enhancing the accuracy of shape detection. Utilising these adaptive thresholding parameters significantly improved the separation of arrows and boxes which addressed the client's requirement for accurate vertex and arrow detection.

- **Morphological Operations:** Morphological operations encompass a set of image-processing techniques that manipulate the structure of features within an image. Using OpenCV's `morphologyEx()` function, the team employed erosion and dilation operations to bridge gaps between pixels, smooth edges, and separate interconnected shapes. The application of these functions allowed further enhancement to the accuracy of shape detection and component separation which contributed to an increasingly refined image recognition process. The results are depicted in Figure 5 and Figure 6, below.

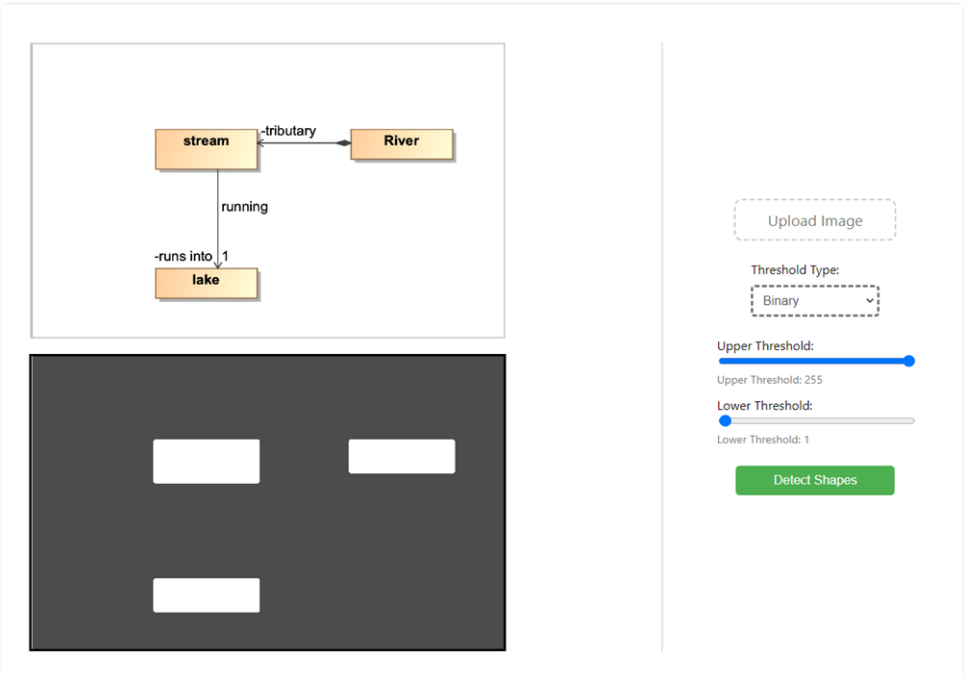

Figure 5: Output Obtained from Applying Adaptive Thresholding and Morphological Operations to a Sample Input.

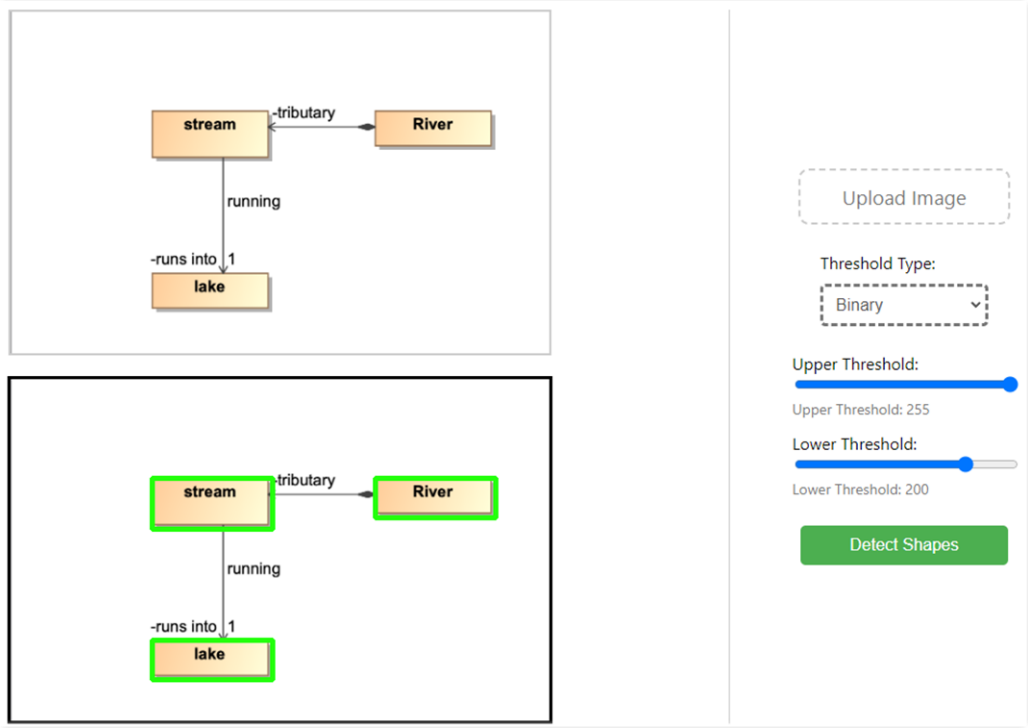

Figure 6: Output Displaying Individual Vertices Separated from a Graph.

2.2 ROADBLOCKS

The path toward seamless image recognition integration encountered several roadblocks that provided valuable insights into the complexities of the task.

Contour Coordination Challenge:

A notable challenge emerged in the accurate extraction of coordinates from detected contours.OpenCV's contour detection functions are often presented as a single entity rather than individual components. This hindered the ability to precisely identify and differentiate multiple shapes within an image which obstructed the accuracy of our shape recognition process. The team identified that overcoming this roadblock was pivotal to the success of the project and was relevant for isolating vertices and their connections.

Vertex-Arrow Detection Conflict:

A significant roadblock arose from the methodology employed for vertex detection. Although the team's approach effectively identified vertices within images, it inadvertently hindered the detection of connecting arrows. The method's focus on vertex identification overshadowed the need for accurate arrow recognition. The result was a partial representation of the graph. This was a critical issue as the client's requirements included the accurate depiction of vertex connections. This challenge emphasized the importance of striking a balance between vertex and arrow detection methods to provide a comprehensive and accurate representation.

Text Recognition Challenge:

Text recognition presented difficulties in isolation and identification. The team's existing approach struggled to differentiate text elements which affected the overall utility of the image recognition module. Addressing this challenge became essential. Not only for providing a holistic understanding of the graph but also for fulfilling the client's requirements for a comprehensive analysis tool.

2.3 RECOMMENDATIONS

Phase 1 was instrumental in gaining a deeper understanding of the challenges associated with integrating image recognition capabilities into the existing graph editor. The experiences and lessons learned in this phase significantly shaped our approach and methodologies as we progressed into subsequent phases of the project.

Leveraging the Prototype's Success:

The successful creation of a prototype that demonstrated OpenCV's shape detection capabilities provided a valuable foundation to expand on. The prototype's ability to detect shapes provided a strong starting point for the development in subsequent phases. The team built on this success to enhance the accuracy and robustness of shape detection with the aim of extending this capability to increasingly complex graph models.

Adapting the Approach for Complex Graph Models:

The challenges encountered when tweaking the application to detect intricate graph models emphasized the need for adaptability. The complexity of the graph models necessitated a shift in strategy, specifically in separating edges and vertices effectively. The successful separation of adaptive thresholding and morphological operations will be applied and refined in future phases, ensuring accurate graph representations are produced.

Addressing Information Extraction Challenges:

The challenge in extracting information from detected objects due to the lack of OpenCV + JavaScript resources was a critical obstacle that needed to be overcome. In subsequent iterations, the team explored innovative strategies and integrated external libraries and tools to enhance information extraction capabilities. This involved researching and implementing methods to effectively parse and interpret the data obtained from shape recognition.

Exploring OpenCV with Python for Enhanced Documentation and Resources:

The team found that comprehensive documentation was available for OpenCV and Python. The shift to OpenCV with Python in Phase 2 stemmed from this realization which enabled a deeper understanding of the required functionality of the application and enhanced the team's ability to extract information from detected objects. This migration empowered the team to tap into a broader knowledge base, fostering efficiency and innovation in the project.

3. PHASE 2 - OPENCV & PYTHON

In Phase 2 of the project, the team transitioned from using OpenCV with JavaScript to OpenCV with Python based on the insights and recommendations from Phase 1. The primary goal remained consistent: to integrate an image recognition module into the existing graph editor, focusing on refining functionality based on the lessons learned in the previous phase. Notable achievements included the improved application of contours to the client's graph models, enhanced detection capabilities, and successfully identifying text elements and connecting arrows. However, challenges persisted in accurate vertex detection and effective information extraction from the detected contours, highlighting the need for further advancements.

3.1 FUNCTIONALITY ACHIEVED

Improved Contour Application:

Phase 2 achieved an enhanced application of contours to the client's graph models. The contours accurately represented the shapes in the graph, providing a more refined representation of individual graph concepts. The contoured image was extracted into a separate file, facilitating easier analysis and manipulation of the graph's structure.

Enhanced Detection Capabilities:

Unlike Phase 1, the application demonstrated improved detection capabilities. It successfully identified and picked up text elements and connecting arrows within the images, enriching the information extracted from the graph. This advancement marked a significant step towards a more comprehensive graph analysis tool. As an example, when using Figure 7 as the input, the application was able to detect relevant graphical data including texts and edges. The final output showcasing this detection can be observed in Figure 8.

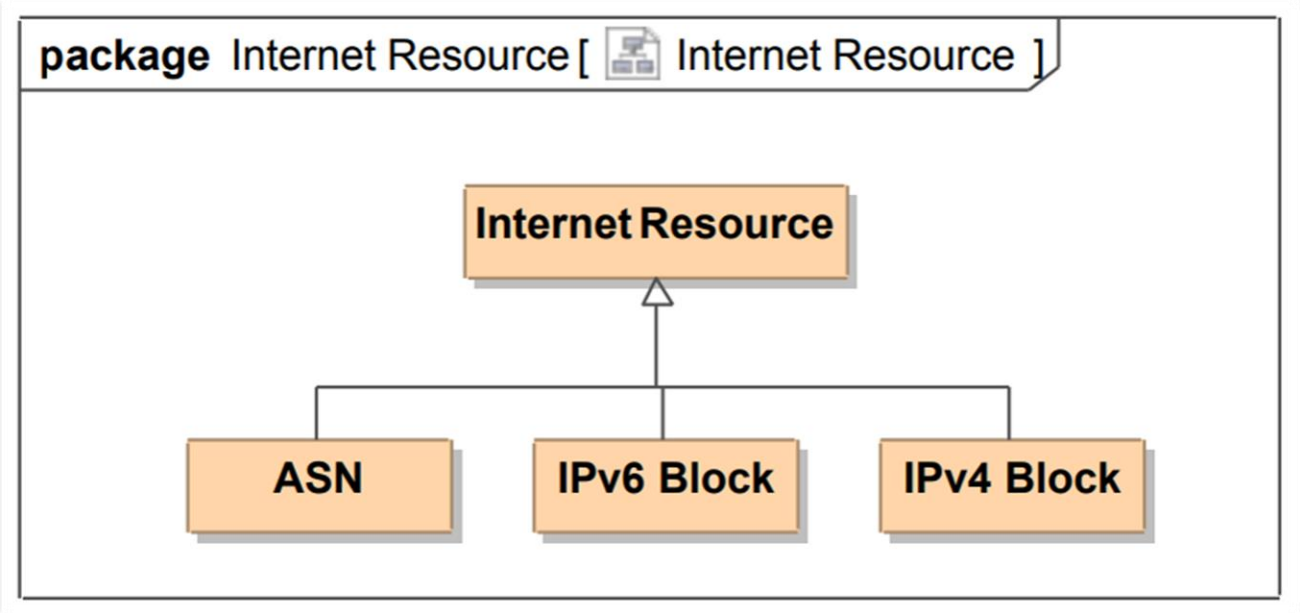

Figure 7: Graph Model Created by S23M, Intended for use as a Sample Input.

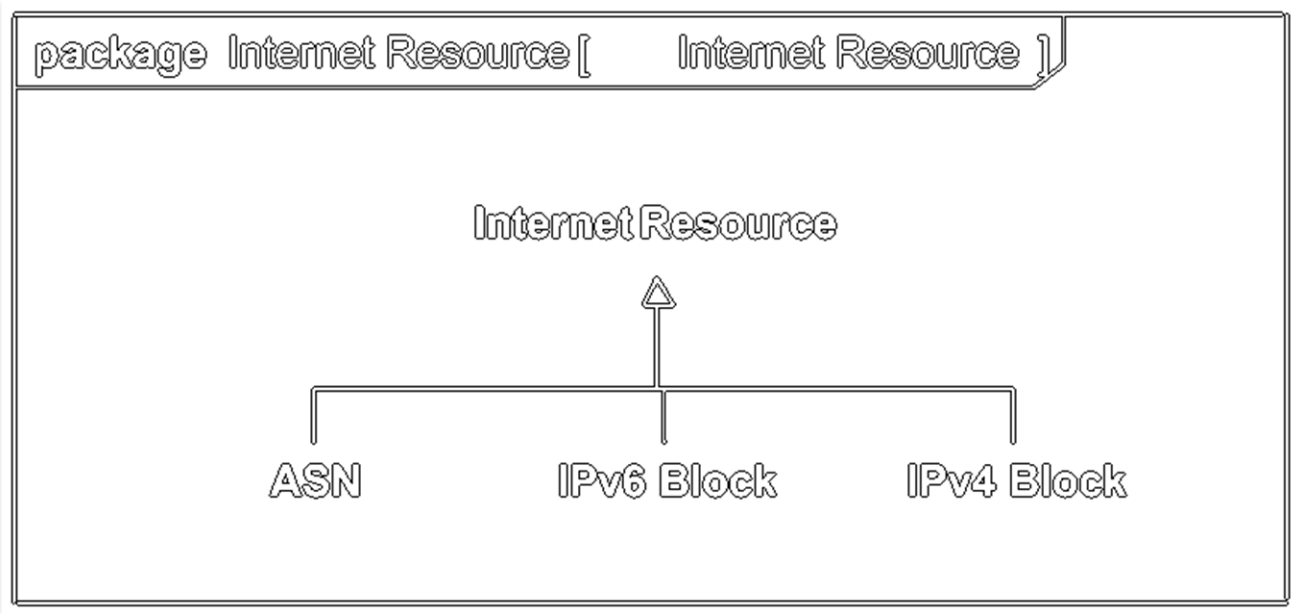

Figure 8: Output Displaying Detected Texts and Edges.

3.2 ROADBLOCKS

Challenges in Vertex Detection:

The challenges persisted when attempting to detect varying degrees of graph complexity, a recurring obstacle from Phase 1. Despite improvements in detecting lines and text, accurately recognizing vertices remained elusive for complex graphs. Overcoming this challenge is crucial for achieving complete and accurate graph representation, aligning with the project's objectives.

Information Extraction Hurdles:

Effectively extracting information from detected contours persisted into Phase 2. This challenge underscored the need for a robust approach to interpret and utilise the detected contours effectively. Overcoming these hurdles will be vital for the successful integration of the image recognition module within the client's model graph editor.

3.3 RECOMMENDATIONS

The experiences and challenges encountered during Phase 2 induced a shift in the team’s approach for the subsequent phase. Despite improvements in contour application and detection capabilities, persistent challenges remained pertaining to accurate vertex detection and effective data structure extraction.

Exploring Custom Image Recognition Training:

Given the recurring challenges of accurately capturing all elements from client models, the team has chosen to explore custom image recognition training. This involves exploring the field of machine learning to train our own image recognition models. By creating tailored models, the team’s aim is to enhance the accuracy and precision of element detection. This includes vertices, edges, text, and connecting arrows. This approach may produce an increasingly customized solution to suit the intricacies of the project requirements which potentially underscores the team’s commitment to finding innovative solutions and leveraging advanced technologies to overcome persistent challenges. The endeavour to develop custom image recognition models aligns with our overarching objective of delivering a robust and precise image recognition module for the graph editor.

4. PHASE 3 - TENSORFLOW & PYTHON

After switching development environments over to Python, it was determined that OpenCV would not be suitable for progressing the project. However, Python yielded better results than JavaScript as it provides a vast amount of external libraries for machine learning and artificial intelligence. After preliminary research was conducted, the popular open-source library 'TensorFlow' was established for the next phase. Instead of the programmatic approach to recognition like OpenCV, TensorFlow allowed a deep neural network model to be trained from the ground up which would be used to run inference on images.

The ultimate goal of the image recognition software is to recognise objects and replicate the detected structure in S23M's model editor; it is the latter part that proves to be the most challenging. It is here we have documented our steps and attempts at importing the recognised objects into the model editor. While the process of translating TensorFlow's output to the editor has not been fully implemented, the team have noted our findings and current research in this section.

4.1 FUNCTIONALITY ACHIEVED

A model was successfully trained on a dataset of computer-generated diagrams produced in both the existing model editor and MagicDraw. The newly trained model has been included as an artefact and is ready to run inference on images using the plot_object_detection_checkpoint.py script. Resources and procedures have also been outlined for training/retraining a model. It should be noted this is a resource-intensive operation. Additionally, the trained model produced results with far greater reliability and accuracy than contour detection with OpenCV and is the recommended method for object detection moving forward. However, this method does introduce additional roadblocks that require further investigation.

Training the Model

Environment:

The first step was to set up the development environment for TensorFlow and model training. This involved installing Python and any relevant packages. With the assistance of the Object Detection API guide (link) and the TensorFlow (link) documentation, scripts were created to begin training a model to detect objects.

Dataset and Partitioning:

Because the goal was to detect computer-generated graphs created in specific drawing software, the model needed to be trained on a custom dataset. The SSD ResNet50 V1 FPN 640x40 model (link) based on the Coco17 dataset was used as a base for the new model to be trained on relevant images. The dataset consisted of graphs that were created in S23M's model editor, with a partitioning of 9:1. This allowed the team to determine the accuracy of the model and track its learning progression against images that had not been recognised previously.

Labelling and Annotating:

The next step was to label the training images. This involved using a program to manually document objects found in images, as shown in Figure 9. These images are what the model examines and are used to instruct the model about the visual representations of objects. For this prototype, 'labelImg' (link) was used. However, any data annotating software is acceptable as long as it stores the annotations in XML files.

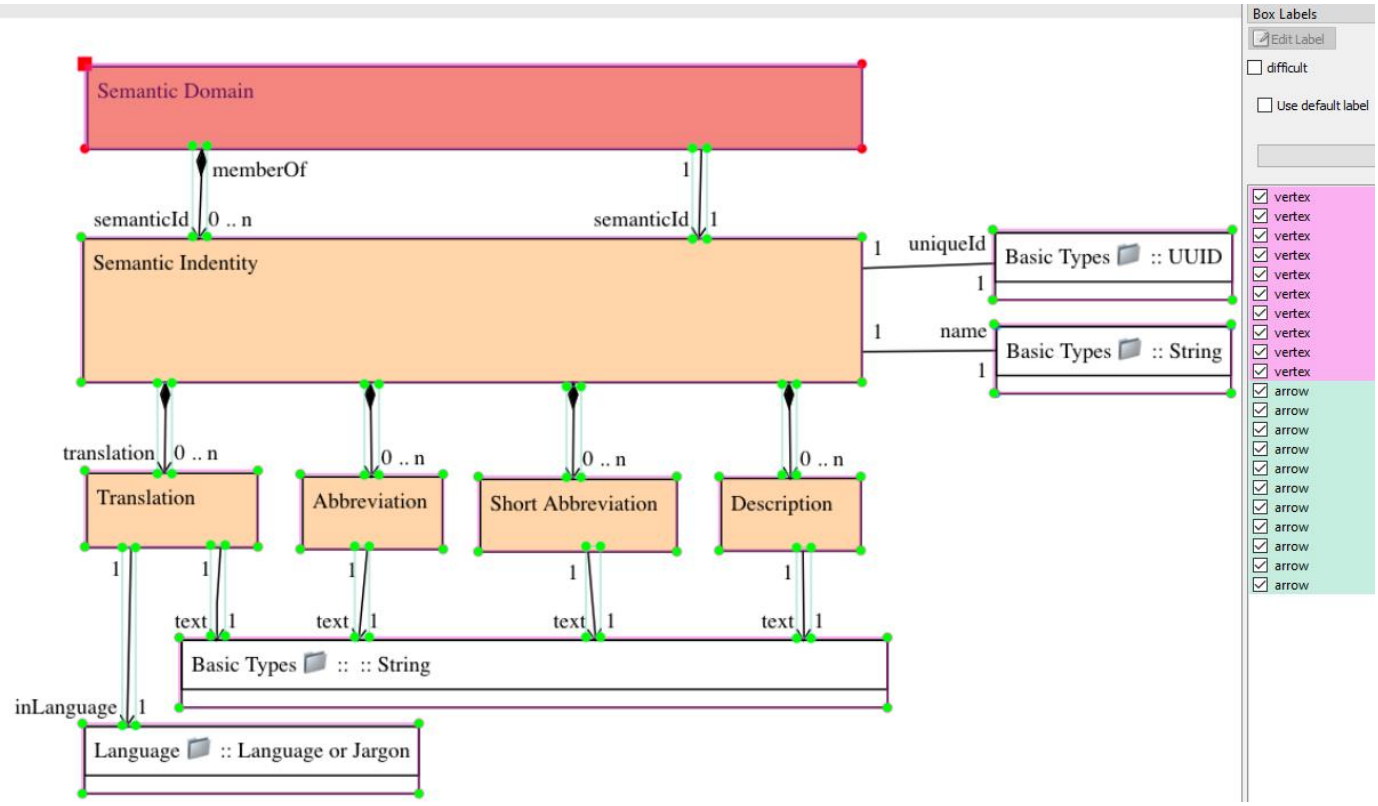

Figure 9: 'labelImg' Annotation on Sample Image.

Label Mapping:

After the images are labelled, TensorFlow requires a label map. This is a .pbtxt file that maps annotations to integers and is required for both the training and detection operations. This contains all of the annotations from the previous step and has a non-negative unique number associated with each one. Each annotation is inside an `item` object with both an `id` and a `name` property, as seen in Figure 10.

Figure 10: Example .pbtxt File

TensorFlow Record Files:

Once the label map was created, the XML files generated by the annotation software must now be converted to TensorFlow record files. A Python script ‘python generate_tfrecord.py’ was compiled to accomplish this. This operation will require the `pandas` library. The relevant function, shown in Figure 11, takes the path to the images and annotations as arguments and will convert any .XML files to .record files that could then be used by TensorFlow.

```
def xml_to_csv(path):
    # Iterates through all .xml files (generated by labelImg) in
    # a given directory and combines them in a single Pandas
    # dataframe

    xml_list = []
    for xml_file in glob.glob(path + '/*.xml'):
        tree = ET.parse(xml_file)
        root = tree.getroot()
        filename = root.find('filename').text
        width = int(root.find('size').find('width').text)
        height = int(root.find('size').find('height').text)
        for member in root.findall('object'):
            bndbox = member.find('bndbox')
            value = (filename,
                     width,
                     height,
                     member.find('name').text,
                     int(bndbox.find('xmin').text),
                     int(bndbox.find('ymin').text),
                     int(bndbox.find('xmax').text),
                     int(bndbox.find('ymax').text),
                     )
            xml_list.append(value)
    column_name = ['filename', 'width', 'height',
                   'class', 'xmin', 'ymin', 'xmax', 'ymax']
    xml_df = pd.DataFrame(xml_list, columns=column_name)
    return xml_df
```

Figure 11: Excerpt from generate_tfrecord.py

Pipeline Config:

Next, the pipeline needed to be configured. This will work regardless of the base model exploited. However, the team utilised the SSD Resnet 50 V1 FPN model. An additional `pipeline.config` file was compiled that needed to be configured. Particular fields of interest are:

- num_classes: Set this to the number of unique annotations present in the label map.
- batch_size: The number of images to process per iteration. Higher numbers will use considerably more memory.
- fine_tune_checkpoint: This needed to be set to the ‘ckpt-0’ folder of the base model.
- fine_tune_checkpoint_type: This parameter needs to be set to `detection`.
- use_bfloat16: This parameter needs to be set to False if the current developer is not training on a Tensor Processing Unit (TPU).
- label_map_path: Path to the .pbtext label map file.

Running Training Job:

Once the pipeline config had been configured, the model was ready for training. Training can be initiated with the 'model_main_tf2.py script' which will take both the model folder and the pipeline configuration file as arguments, as shown in Figure 12. This began training a new model. Note that this process is resource-intensive and will be *significantly* faster on a discrete graphics processing unit, which may still take hours depending on the dataset size. The prototype model was trained on an NVIDIA RTX 2060 GPU and took approximately 3-4 hours. Once training had been completed, there were generated checkpoint files in the new model folder which could now be used to run inference on images.

```
python model_main_tf2.py \
--model_dir=models/my_ssd_resnet50_v1_fpn \
--pipeline_config_path=models/my_ssd_resnet50_v1_fpn/pipeline.config
```

Figure 12: model_main_tf2.py invoked with arguments

Running Inference:

When running inference against previously undetected images, the trained model had a high degree of accuracy. This was likely due to the simplicity of the objects the model was required to recognise. However, complex graphs induce less accurate results. This can be improved over time by retraining the model with additional data and feeding the model comprehensively labelled complex graphs. Relative to OpenCV, this method produced progressively reliable outcomes and didn't encounter the limitations of contours. However, it can produce inaccurate data, as shown in Figure 13. Figure 14 displays the refined model, showcasing the model's ability to produce increasingly accurate results.

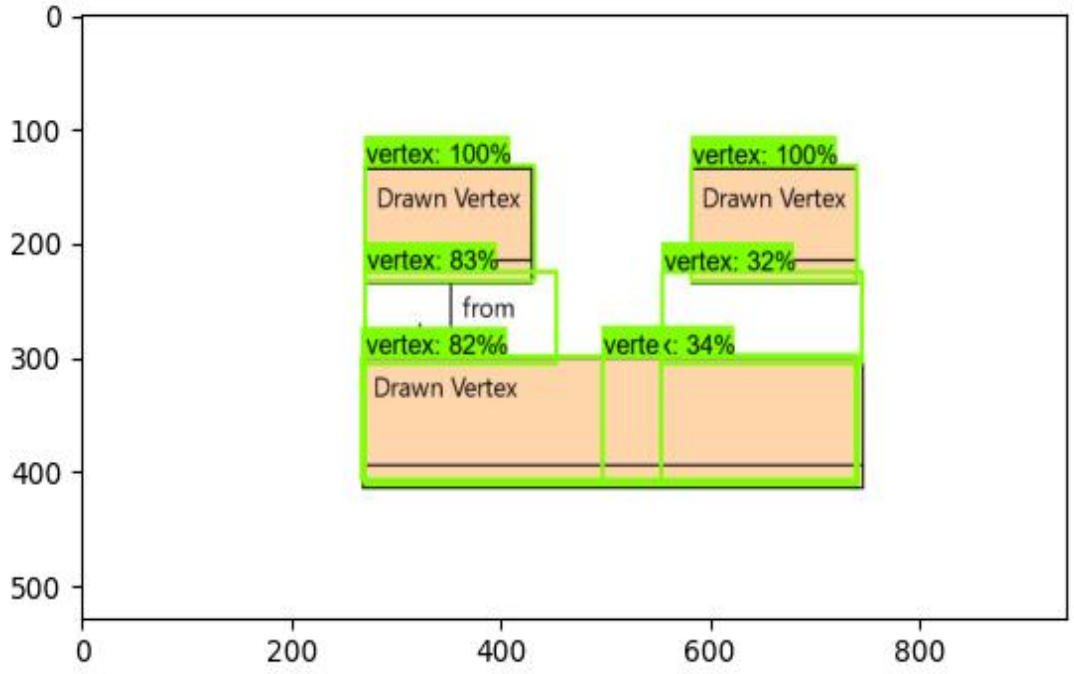

Figure 13: Output from Initial Model

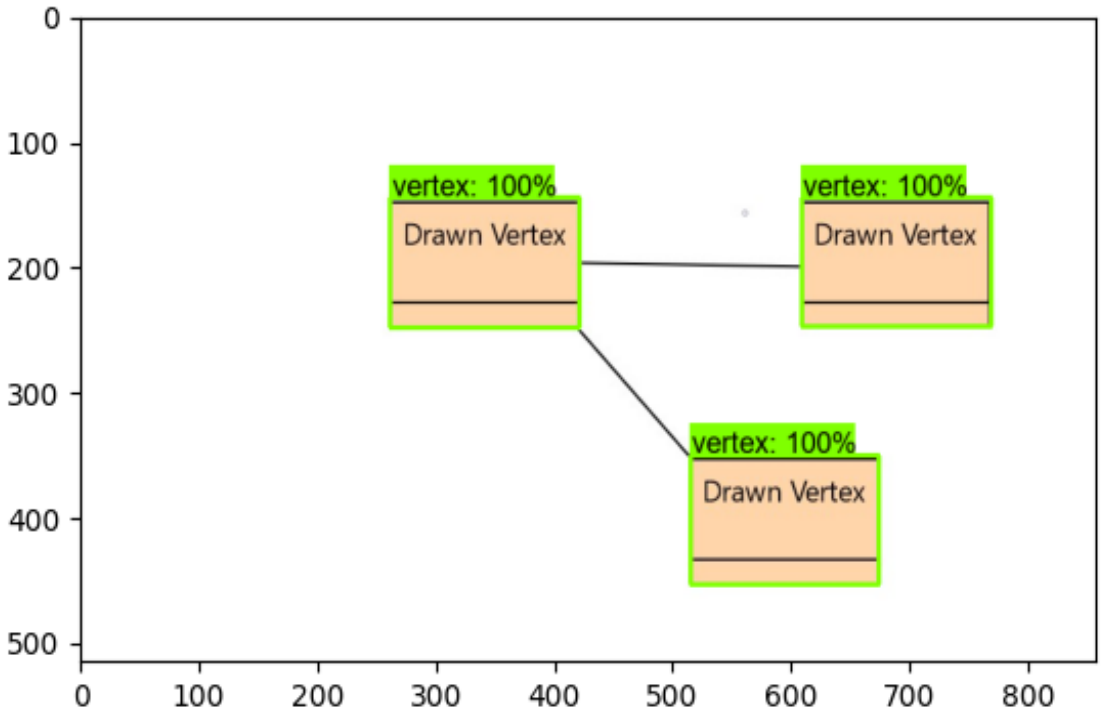

Figure 14: Output from Refined Model

4.2 ROADBLOCKS

The primary roadblock of this method of object detection isn't the object detection itself as the current prototype runs detections, stores them in memory, overlays them onto the image, and saves the result to an external file. While this is an improvement over previous iterations, the ultimate goal is to have the detected objects extracted to S23M's web-based editor and accurately replicated. This process introduced considerable complexity to project progression, as TensorFlow is a large and complicated library that abstracts elements of the detection process. The next logical step was to intercept the detection method to obtain a usable coordinate structure to extrapolate to model editor coordinates. The team found that the inference script stores the detections in an appropriately named variable called `detections`, as seen in Figure 15 below. The detected objects are obtained by running the `predict` method of the object detection API model builder library.

```
def detect_fn(image):
    # Detect objects in image

    image, shapes = detection_model.preprocess(image)
    prediction_dict = detection_model.predict(image, shapes)
    detections = detection_model.postprocess(prediction_dict, shapes)

    return detections
```

Figure 15: Python function for Object Detection

The 'detections' variable is a dictionary containing numerous other dictionaries, arrays, and values. Objects of interest are the `detection_boxes`, `detection_classes`, and `detection_scores` dictionaries. Stepping through with a debugger and inspecting 'detection_boxes' revealed an array of coordinates. However, they are in sets of 4 and there are 400 sets, as shown in Figure 16.

```
detections: {'detection_boxes': array([[0.28256148, ...e=float32), 'detection_scores': array([[
> special variables
> function variables
> 'detection_boxes': array([[0.28256148, 0.71014965, 0.4811625 , 0.8976135 ],
> 'detection_scores': array([0.9999912 , 0.99998546, 0.9999734 , 0.02406907, 0.01866413,
> 'detection_classes': array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

Figure 16: Data Structures in Memory

Upon closer investigation, the sets of 4 values are ‘ymin’, ‘xmin’, ‘ymax’, and ‘xmax’ respectively. Given the model normalises the image during the preprocessing stage, the team has concluded that each value may be representative of the location of each pixel of interest. Each array is essentially a tensor that represents the vector space of the two-dimensional image or object that is supplied to the model. These arrays may be comprised of normalised bounding box coordinates that are potentially in unit vector form. The size and length of the detected objects may have to be scaled appropriately using unit vector coordinates in order to extrapolate the required information from the model’s response to S23M’s model editor’s relative coordinate system. Additional experimentation is required to ensure accurate results are obtained.

4.3 RECOMMENDATIONS

Overall, despite the roadblocks, it appears Python and TensorFlow produce a high degree of validity and reliability in detecting required objects. Utilising OpenCV for contour extrapolation becomes increasingly complex and prone to inaccuracies as the complexity of the task grows. Enhanced precision and robustness can be attained by training a custom TensorFlow model on a dataset tailored to the specific recognition requirements.

Currently, the inference script displays results on a ‘matplotlib’ graph and saves it as an image file in the working directory. This can be substituted with an alternative method, provided that the substitution occurs after the model has completed its analysis of the image and holds the detection results in memory. The code to plot detections and export to an image is displayed in Figure 17, below. Additionally, the values for loading the model for inference are currently hardcoded at the script outset. These values may be dynamically loaded through programmatic means, such as utilising a graphical user interface or command-line interface (CLI) arguments. The code has been commented with various observations, reasonings, and test code that has been preserved for potential future endeavours aimed at advancing this project. For an unknown reason, inference has been unsuccessful on PNG image files. However, JPG files appear to function without issue.

```
import matplotlib.pyplot as plt

plt.figure()
plt.imshow(image_np_with_detections)
print('Done')

plt.show()
plt.savefig("output.png")
sys.stdout.flush()
```

Figure 17: Code to Plot Detections and Export to Image

It is recommended to step through the program in debug mode and inspect memory values as it executes, most notably after the detections have been loaded. Inspecting the `detections` variable at this point will reveal the data structures relevant to extracting coordinates and any other related data. Figure 18 shows the relevant breakpoint section.

```
145    viz_utils.visualize_boxes_and_labels_on_image_array(
146            image_np_with_detections,
147            detections['detection_boxes'],
148            detections['detection_classes']+label_id_offset,
149            detections['detection_scores'],
150            category_index,
151            use_normalized_coordinates=True,
152            max_boxes_to_draw=200,
153            min_score_thresh=.30,
154            agnostic_mode=False)
```

Figure 18: Relevant Debug Detections for Future Developers.

5. FUTURE RECOMMENDATIONS AND CONSIDERATIONS

The culmination of Phase 1, Phase 2, and Phase 3 establishes a solid framework to expand on in further iterations. This section provides guidance and considerations for future developers, offering a roadmap to navigate the intricacies of this project successfully.

Developers considering OpenCV and Python/JavaScript should consider the following recommendations:

Refining Vertex and Edge Detection:
Future developers continuing from Phase 1 and Phase 2 should explore methods to separate vertices and edges from detected contours effectively. One approach could involve running separate processes to detect vertices, edges, and texts using optical character recognition. The results from these processes can then be combined to create a comprehensive JSON mapping structure. Building on the effectiveness of adaptive thresholding and morphological operations observed in Phase 1, efforts should be directed towards improving the accuracy of morphological operations. Experimentation with erosion and dilation operation values can yield better results which may address technical challenges such as grouping the region between two edges as a single vertex due to resemblance to a square.

Leveraging Contour Hierarchy for Data Extraction:
Future developers should delve into understanding OpenCV's contour hierarchy, a concept identified from Phase 1. Utilising contour hierarchy can provide better control over the results, enabling targeted data extraction from detected contours. It offers the potential to focus on specific detected objects and obtain relevant graphical data, contributing to increasingly accurate data extraction.

Optimising Development Environment:
Based on the team’s experience, future developers are encouraged to prioritize using Python over JavaScript for OpenCV. Python offers a wealth of accessible resources and facilitates the development of a test environment efficiently. While JavaScript was initially chosen due to the client's model editor being based on ReactJS, transitioning to Python aligns with the abundance of resources and ease of development associated with the language.

Developing a Bridge Program:

An alternative approach involves creating a program to bridge between the graph editor and the image detection module. This program could automate the creation of vertices in the editor based on the detection count from the image module. However, future developers should be aware that manual adjustments may still be necessary for accurate positioning and sizing of vertices in the editor. Additionally, they should consider refactoring code, implementing functions to automate object creation, and leveraging object-oriented programming principles to enhance efficiency.

Developers expanding on the current development cycle of Tensorflow and Python should consider the following recommendations to achieve desired outcomes:

In-depth Research on Custom Image Recognition:

Future developers should embark on a comprehensive exploration of custom image recognition. This entails diving into machine learning concepts, particularly in the domain of image recognition models. Understanding various algorithms, architectures, and training techniques is essential. Research should focus on prevalent frameworks like TensorFlow, PyTorch, and scikit-learn, enabling informed decisions on the most suitable approach for training custom models tailored to the project's needs.

Exploration of Transfer Learning:

Transfer learning stands as a powerful tool in the realm of image recognition. Future developers should extensively research and experiment with transfer learning techniques. By leveraging pre-trained models and adapting them to suit the specifics of the graph analysis, considerable time and computational resources can be saved. Understanding how to fine-tune these models for accurate vertex, edge, and text detection will be instrumental.

Integration of Neural Network Architecture

Given the complexity of the task, employing neural network architectures, such as convolutional neural networks (CNNs), should be a focal point. Understanding how CNNs can be structured and optimized for the graph analysis task will be crucial. Researchers should explore CNN variants and adapt them to accurately detect various components within a model, ensuring a comprehensive representation.

Investigation into Data Augmentation Techniques

Data augmentation plays a pivotal role in enhancing the robustness of image recognition models. Future developers should thoroughly investigate diverse data augmentation techniques, including rotations, translations, scaling, and brightness adjustments. Implementing these techniques effectively can significantly improve model performance by providing a diverse and enriched dataset for training.

Collaboration with Domain Experts

To enhance the accuracy and relevance of the image recognition module, collaboration with domain experts in graph theory or related fields is highly recommended. Their insights can guide the development process, ensuring that image recognition accurately captures the nuances of graph structures and components.

Absolute versus Relative Coordinate Values

To ensure the spacing and size of each detected object are transferred successfully to S23M's model editor application, the relative coordinates from the trained image and model editor need to be considered. Future developers may produce inaccurate results when implementing absolute values for coordinate systems. Relative values may be implemented by producing a mathematical algorithm that normalises canvas coordinates relative to external images.