DOKUMENTATION

PROJEKT 1

Danielle Ndjensi & Joel Ngounouo Kamga

1. Zielsetzung und Umfang des Projekts

Das Hauptziel des Projekts ist es, verschiedene Prozesse rund um handgezeichnete Graphen zu untersuchen. Die Arbeit umfasst folgende Schwerpunkte:

- Graphen selbst zeichnen, optimieren und labeln
- Erkennen von handgezeichneten Graphen
    - Knoten und Kanten
- Erstellen eines Graphen-Models
- Graphen in „schön“ zeichnen

2. Datenerfassung

Ein entscheidender Schritt bei der Entwicklung eines leistungsfähigen Lernmodells ist die Datenerfassung. Hier sollte man **einen Datensatz aus Bildern erstellen.** Eine qualitativ hochwertige und strukturierte Datengrundlage ist essenziell für das Training und die Optimierung des Modells. Deswegen haben wir die folgenden Anforderungen identifiziert:

- handgezeichneten schwarzen Graphen
- Graphen mit mindestens 15 Knoten
- Graphen mit unterschiedlichen Merkmalen und Strukturen wie Bäume, planare Graphen und Graphen mit Kantenüberkreuzungen
- Die Knoten der Graphen sollten hierbei gelabelt sein
- sauber und präzise gezeichnete Graphen sowie weniger präzise und unordentlich gezeichnete Graphen

3. Graph Annotation

Annotation von Graphen aus dem Datensatz ist für das Training des Modells zur Erkennung von handgezeichneten Graphen unerlässlich. Man sollte dem Modell zeigen, wie Knoten und Kanten eines Graphen aussehen, damit es auch lernt. Dafür haben wir das Tool LabelStudio verwendet.

3.1. Kürzer Überblick in LabelStudio

LabelStudio ist ein Open-Source-Tool zur **Datenannotation**, das für maschinelles Lernen und KI-Modelle verwendet wird. Es ermöglicht das **Labeln** (manuelle Markieren) von Daten in verschiedenen Formaten, darunter Bilder**,** Texte, Video, …. Wir haben LabelStudio hier als Annotationswerkzeug verwendet, um die Knoten und Kanten in den handgezeichneten Graphen zu markieren.

3.2. Vorbereitung des Labeling Interface

Bevor man mit der Annotation begonnen werden kann, muss die Labeling-Oberfläche entsprechend eingerichtet werden, um eine effiziente und präzise Kennzeichnung der Graphenelemente zu ermöglichen. Zum makieren von Kanten und Knoten haben wir uns für „**Object detection with Boundind Boxes“** entschieden. Danach haben wir die zwei Klassen definiert: edge(für Kante) und node (für Knoten)

3.3. Annotation

Hier wird die verwendete Methodik beschrieben:

- Präzise und enganliegende Bounding Boxes: Die Box umfasst genau die Knoten und Kanten mit minimalem Abstand
- Richtige Handhabung von Überkreuzungen
- Vermeidung oder Minimierung von Überlappungen von Boxes

3.4. Exportieren des Projekts

Nachdem alle Graphen unserem Datensatz annotiert wurden, sollte man das Projekt exportieren, damit wir die Bilder und ihre entsprechenden Annotationen sammeln können. Dafür sollte man das Projekt in YOLOv8-Format exportieren, denn das Modell wird mithilfe von YOLO trainieren (siehe 6.)

Beschreibung des exportierten Projekts

Das exportierte Projekt enthält folgende Dateien:

- ***images* -Ordner :** Enthält die Bilddateien, die wir annotiert haben
- **labels- Ordner** : Enthält die Annotationen (die **Bounding Box Labels** im YOLO-Format) von jedem Bild. Jedes Bild hat eine zugehörige ***.txt***-Datei mit denselben Namen. Jede Zeile beschreibt eine Bounding Box im Format: **class_id x_center y_center width height,** wobei **class_id** die Klasse des Objekts ist, **x_center** und **y_center** die Mitte der Bounding Box beschreiben, und **width** und **height** die Breite und Höhe der Box beschreiben.

4. Bildvorverarbeitung

Bevor wir mit dem Training anfangen, sollte man sicherstellen, dass die Daten sauber und bereinigt sind. Das Ziel ist es, die Qualität der Eingabedaten verbessern, um die Erkennung von Knoten und Kanten zu erleichtern**.** Die wichtigsten Aufgaben sind hierbei folgende:

- **Verbesserung der Modellgenauigkeit durch Rauschreduzierung**: Man hat festgestellt, dass ohne Rauschreduzierung das Modell bei manchen Graphen kleine fehlerhafte Punkten und Flecken als Knoten und kleine Striche im Bild als Kanten erkannte.
- **Einheitliche Eingangsgröße für das Modell**: Da neuronale Netze mit festen Tensorgrößen arbeiten, soll sich das Modell vor der Verarbeitung der Daten jedes Mal anpassen, wenn die Eingangsgröße variiert. Man kann sagen, dass verschiedenen Eingangsgrößen führt zu Inkonsistenzen. Wir hatten zunächst versucht, mit unterschiedlichen Eingangsgrößen zu trainieren und haben festgestellt, dass die erkannten Bounding Boxen bei der Erkennung von Knoten und Kanten häufig nicht so exakt und richtig platziert.
- **Normalisierung der Pixelwerte**: Die Trainingszeit, die lang war, wurde durch die Normalisierung der Pixelwerte reduziert.

Für diese Aufgaben haben wir zwei Tools verwendet: Scikit-image und Pillow. Bei verschwommenen, unklaren oder kontrastarmen Bildern haben wir die Erhöhung des Kontrasts von Scikit-image verwendet, um die Strukturen deutlicher hervorzuheben. Bei Bildern mit Rauschen oder unnötigen Details haben wir die Gaußsche-Filterung von Pillow verwendet. Die Größe aller Bilder wird mit Scikit-image auf 416 x 416 gesetzt.

5. Definition der Datenstruktur

Wir haben hier eine Klasse in Python definiert, um die Graphen darzustellen. Die Graph-Klasse enthält zwei Attributen:

- **nodes**: Ein Dictionary, dass dazu dient, die Knoten des Graphen zu speichern. Jeder Schlüssel im Dictionary ist die eindeutige Kennung (node_id) eines Knotens, und der Wert ist ein Tupel (x, y), dass die Koordinaten des Knotens im zweidimensionalen Raum darstellen.
- **edges**: Eine Liste, die die Kanten des Graphen enthält. Jede Kante wird als Tupel (node_id1, node_id2) dargestellt, was eine Kante zwischen dem Knoten node_id1 und dem Knoten node_id2 bedeutet.

6. Training

Das Training des neuronalen Netzes basiert auf den annotierten und vorverarbeiteten Daten. In diesem Abschnitt werden die verschiedenen Schritte des Trainingsprozesses erläutert.

6.1. Wahl des Trainingsverfahrens

Für dieses Projekt haben wir unser Modell nicht von Null trainiert, sondern haben wir ein vortrainiertes Modell zur Objekterkennung benutzt. Hier wird das Ultralytics YOLO mit einem vortrainierten Modell (yolov8n.pt) von YOLOv8 verwendet.

6.2. Organisation der Daten

Die Daten werden in zwei Sets aufgeteilt:

- Trainingsdaten (Training Set, 80%)
- Validierungsdaten (Validation Set, 20%)

Wir haben möglichst versucht, dass die beiden Sets Graphen mit allen Merkmalen und unterschiedlichen Strukturen besitzen.

6.3. Durchführung des Trainings

Nachdem die Trainingsdaten und Validierungsdaten gut organisiert, kann man das Training starten. Wie oben erwähnt haben wir YOLOv8 benutzt. Bevor man das Training startet, soll man die YAML-Datei konfigurieren, damit das Modell weiß, wo sich die Daten befinden, und was sind die Klassen, die man erkennen möchte. Vor dem Start des Trainings ist zudem die Anpassung der Hyperparameter erforderlich. Bei der ersten Durchführung haben wir die Standardwerte der Hyperparameter belassen. Da wir nicht so viel Menge an Daten wird die Anzahl der Iterationen hier auf 100 Epochen gesetzt.

6.4. Evaluation des Modells

Wir haben mit unterschiedlichen Parametern unser Modell trainiert und jedes Mal die Metriken und die erkannten Graphen untersucht:

- Am Anfang mit allen **Hyperparameter auf Standardwerte** und Anzahl von Epochen = 100: Beim Trainieren hat man festgestellt, dass es ab ungefähr der 30. Iteration keine Verbesserung der Validierungsmetriken (val_loss) gab. Am Ende des Trainings hat das Modell Overfitting gezeigt. Es hatte auf den Trainingsdaten eine sehr hohe Genauigkeit

von 0.98 aber beim Testen auf einem Graphen erkannte das Modell mehr Boxen als gewünscht (siehe Bild 1 und 2).

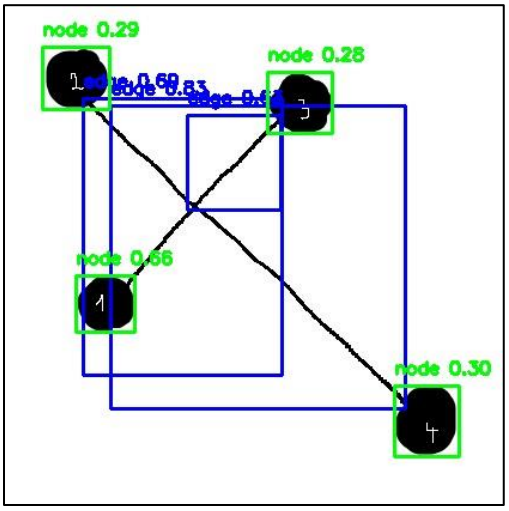


Bild 1 : Graph Erkennung 1

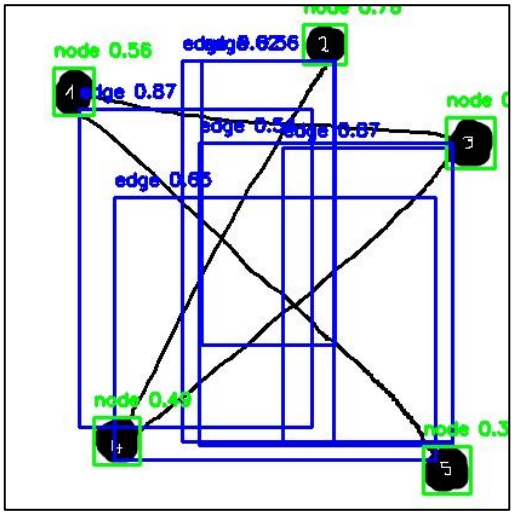


Bild 2: Graph Erkennung 2

- Die Anzahl von Epochen wurde auf 50 reduziert und die Learning Rate auf 0.001 gesetzt: Da das Modell Overfitting zeigte, haben wir die Anzahl von Epochen und die Learning Rate(standard auf 0.1 gesetzt ist) modifiziert. Während des Trainings haben wir nochmal festgestellt, dass es keine Verbesserung von Training-/Validierungsmetriken gab. Und das Modell zeigte nochmal Overfitting am Ende. Wir dachten dann, dass dies an der geringen Größe unseres Datensatzes lag.

- Die Anzahl von Epochen eps=25: Mit dieser Reduzierung haben wir eine relative Stabilität beim Training. Die Verlustfunktionensind niedrig geblieben: test-loss bei 0.88 und val-loss bei 0.75. Einerseits erkennt das Modell gut die Knoten eines Graphen, anderseits erkennt es weniger gut die Kanten, vor allem bei Kantenüberkreuzungen gibt es manchmal unerwartete Bounding Boxen zurück.

Einige Graph Erkennungen :

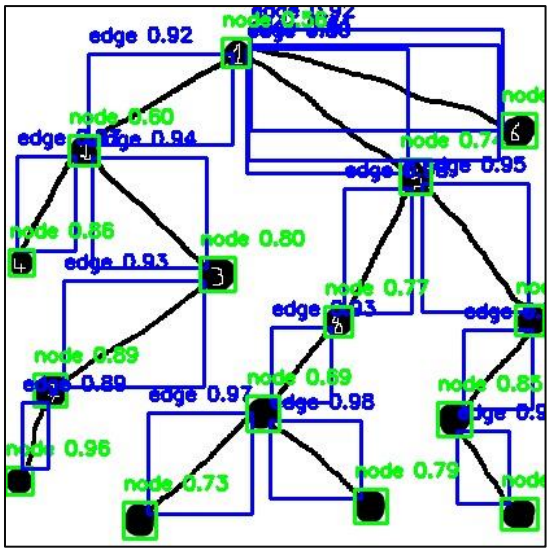


Bild 4: Graph Erkennung (Baum)

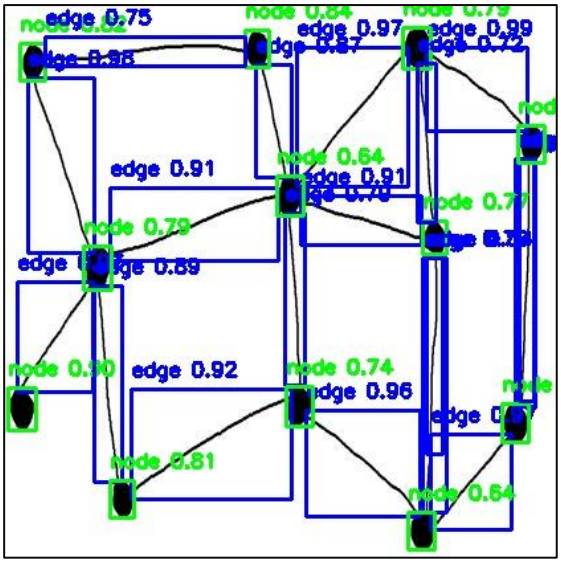
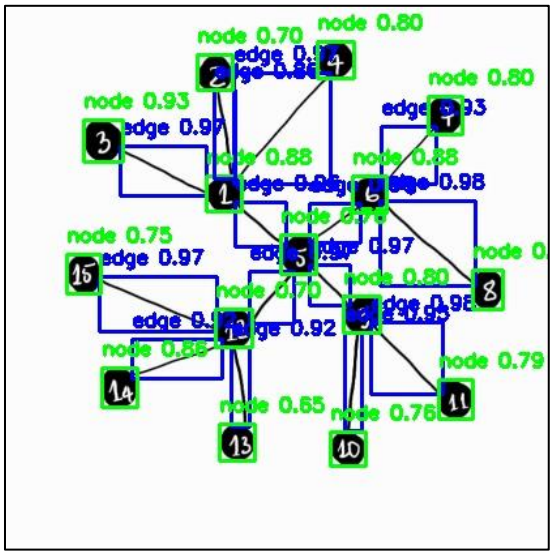


Bild 5 : Graph Erkennung (planarer Graph)



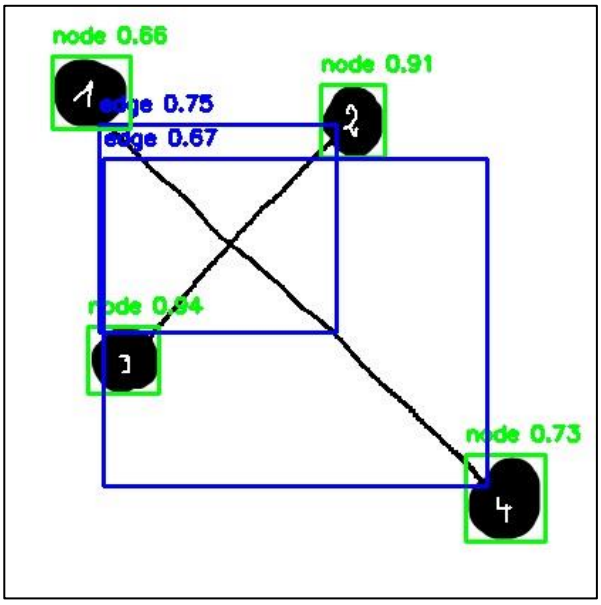


Bild 6: Graph Erkennung (planar) Bild 7: Graph Erkennung (mit Kantenüberkreuzung)

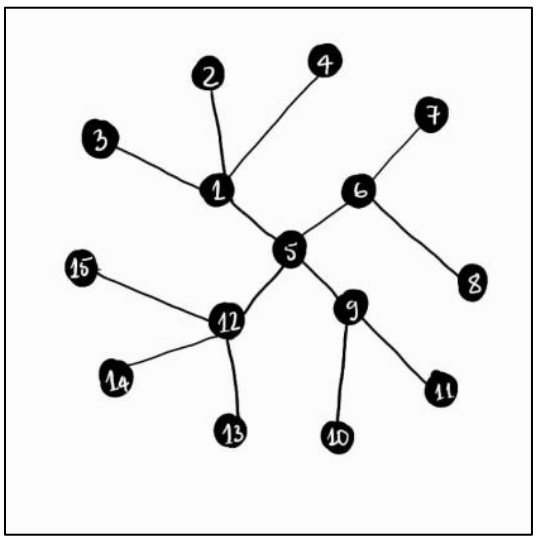
7. Erkennung und Entfernung von Knoten eines Graphs

7.1. Erkennung von Knoten

Für die Erkennung von Knoten haben wir unser trainiertes Modell verwendet. Die Klasse von Knoten wird beim Training mit 1 codiert. Bei der Erkennung selektieren wir nur die Boxen der Klasse 1.

7.2. Entfernung von Knoten aus einem Graph

Nachdem wir die Knoten schon erkannt haben, sollten wir die aus dem Graph entfernt. Dafür haben wir Scikit-Image verwendet. Die Bounding Boxen, die als Knoten erkannt wurden, füllen wir mit der Hintergrundfarbe.



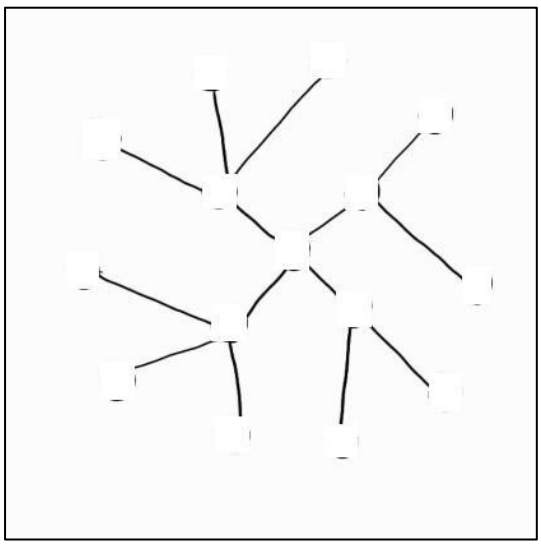


Bild 8: handgezeichneter Graph Bild 9: Graph nach der Entfernung von Knoten

8. Rekonstruktion des Graphen

Nachdem die Knoten erkannt und entfernt wurden, muss der Graph rekonstruiert werden, um eine verwertbare Struktur zu erhalten.

- Zuerst haben wir unser trainiertes Modell verwendet, um die im Bild verbleibenden Kanten zu erkennen (die Bounding Boxen von Klasse 0).
- Die Konstruktion des Graphen erfolgt bei der Zuordnung von Kanten mit Knoten.

8.1. DBSCAN-Clustering

Die hauptsächlich zur Zuordnung verwendete Methode war das DBSCAN-Clustering (Density-Based Spatial Clustering of Applications with Noise).

Diese Methode verwendet als Eingabe eine Reihe von Punkten. Diese Punkte sind die Endpunkte der erkannten Knoten.

- An Anfang erstellen wir ein leeres Dictionnary **nodes** und eine leere Liste **edges,** die wir benutzen werden, um der Graph abzubilden
- Wir speichern alle Endpunkte von Kanten in ein Numpy-Array.
- Danach gruppieren wir mithilfe des DBSCAN die Endpunkte, die nahe beieinander liegen, zu **Knoten**. Hier müssen zwei Parameter übergeben werden: **eps** und **min_samples.** eps ist die maximale Distanz zwischen zwei Punkten, damit sie als Teil desselben Clusters (Knotens) betrachtet werden. **min_samples** wird auf 1 gesetzt, damit jeder Punkt ein Cluster bilden kann, auch wenn er isoliert ist.
- Dann werden die Cluster-Labels der Punkte abgerufen: Jeder Punkt erhält eine Cluster-Nummer (z. B. 0, 1, 2, …) und Punkte mit demselben Label gehören zum gleichen Cluster.
- Nachdem wir unsere Cluster haben, berechnen wir die **Cluster-Zentren** von allen Clustern:
  - Eine Schleife iteriert über alle eindeutigen Cluster-Labels
  - Alle Punkte, die zu einem bestimmten Cluster gehören, werden ermittelt
  - Die **mittlere Position** dieser Punkte wird berechnet (mean(axis=0)), um das Cluster-Zentrum zu bestimmen
  - Dieses Zentrum wird als Knoten-Position in einem Dictionary **nodes** gespeichert. Das bedeutet, am Ende dieses Schritts enthält das Dictionnary die mittleren Positionen der Cluster, die die Knoten des Graphen darstellen
- Zuordnung der Knoten zu den Kanten : wir benutzen nochmal die Endpunkte von erkannten Kanten. Für beide Endpunkten einer Kante **ordnen wir der nächstgelegenen Knoten** aus unseren Clustering-Labeln. Dafür wird die **euklidische Distanz** zwischen jedem Punkt und allen Punkten berechnet. Danach wird der nächstgelegene Punkt innerhalb eines Clusters bestimmt und sein Label zugewiesen
- Hinzufügen der Kante zwischen den beiden Knoten in die Liste, die Kanten speichert

Hinweise: Dieses Clustering basiert auf allen Endpunkten der Kanten. Bei der Erkennung erhalten wir jedoch von unserem Modell nur Bounding Boxen. Deswegen müssen wir noch die Endpunkte der Kanten aus dem Boxen extrahieren. Dafür haben wir den Hough-Line-Detection Algorithmus verwendet.

8.2. Anwendung des Hough-Line-Detection Algorithmus

Das YOLO-Modell erkennt die Kante mit einem Bounding Box. Dieses Feld enthält:

- **x_min**, **y_min**: Koordinaten der oberen linken Ecke des Feldes
- **x_max**, **y_max**: Koordinaten der unteren rechten Ecke des Feldes

Diese Daten beschreiben jedoch nur einen rechteckigen Quader. Deswegen müssen wir die genauen Enden der Kante (bei der es sich um eine Linie handelt) identifizieren. Dabei sind wir folgendermaßen vorgegangen:

- Extrahieren des Bildes oder einer Teilmenge von Pixeln aus dem Bounding Box: Verwendung der Koordinaten des Bounding Box, um den Interessenbereich (ROI) mit der Kante zu beschneiden
- **Hough-Line-Detection Algorithmus** anwenden, um Linien zu detektieren. Eine Dokumentation und eine Implementierung von diesem Algorithmus haben wir vom folgenden Git-repository benutzt: https://github.com/adityaintwala/Hough-Line-Detection
- Die genauen Enden der Kante extrahieren.

8.3. Graph konstruieren

Wie schon in 8.1 erwähnt, am Ende des Clusterings und der Zuordnung von Kanten und Knoten haben wir ein Dictionnary **nodes** und eine Liste **edges.** Mit die beiden können wir ein Graph konstruieren, wie wir unsere Datenstruktur im Abschnitt 5 definiert haben.

Wir haben eine Funktion geschrieben, um den Graphen mit Matplotlib zu plotten.

Einige Implementierungen:

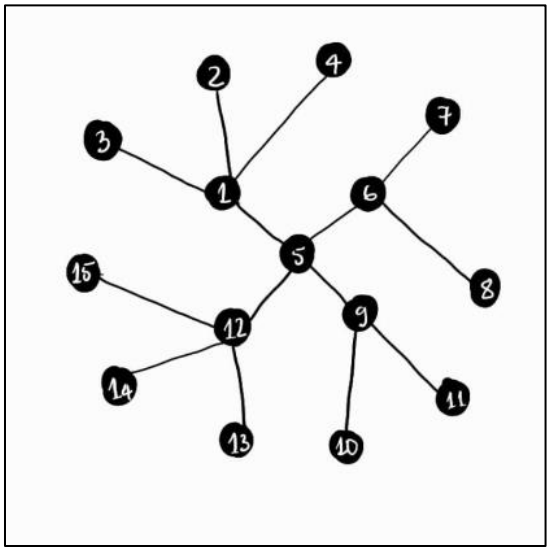


Bild 10: Eingabebild 1

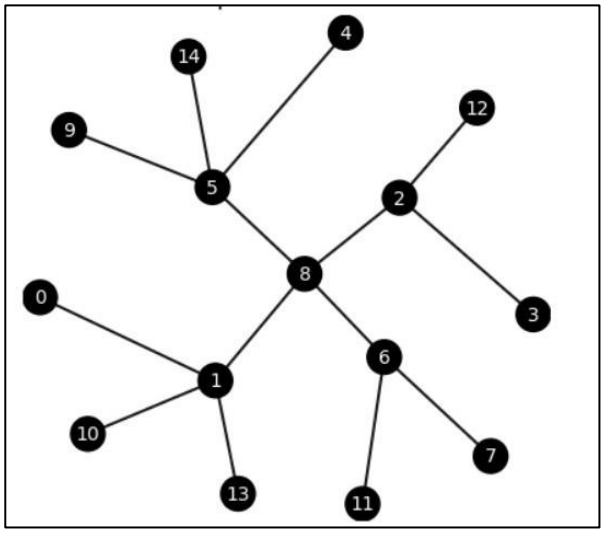


Bild 11: Ergebnis beim Plotten des konstruierten Graphen

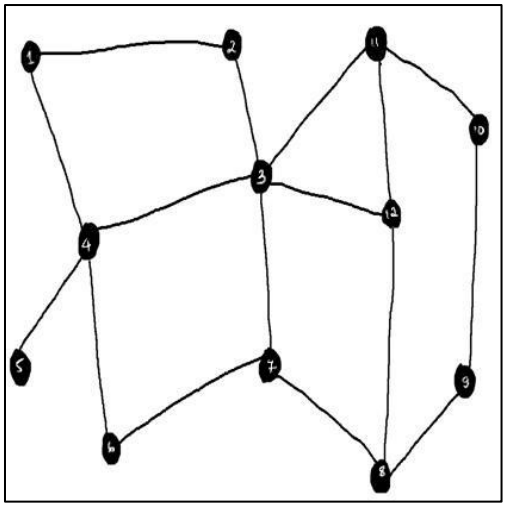


Bild 12: Eingabebild 2

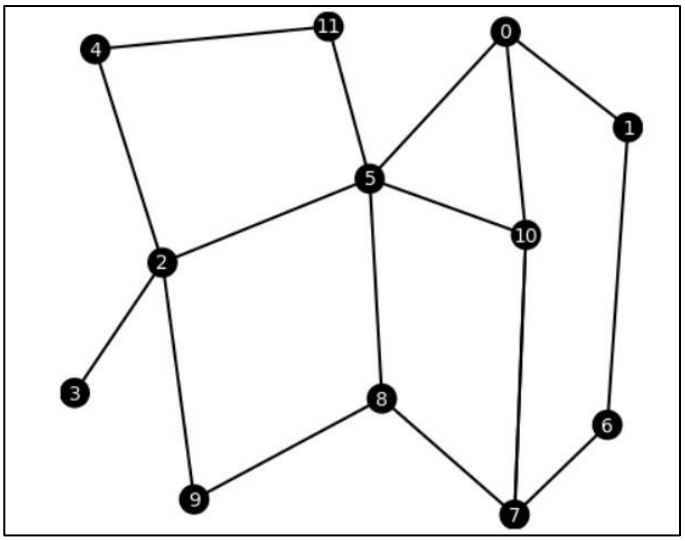


Bild 13: Ergebnis beim Plotten des konstruierten Graphen

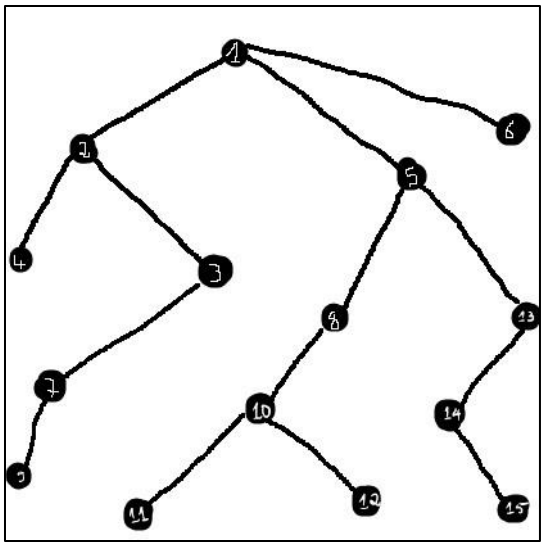


Bild 14: Eingabebild 3(Baum)

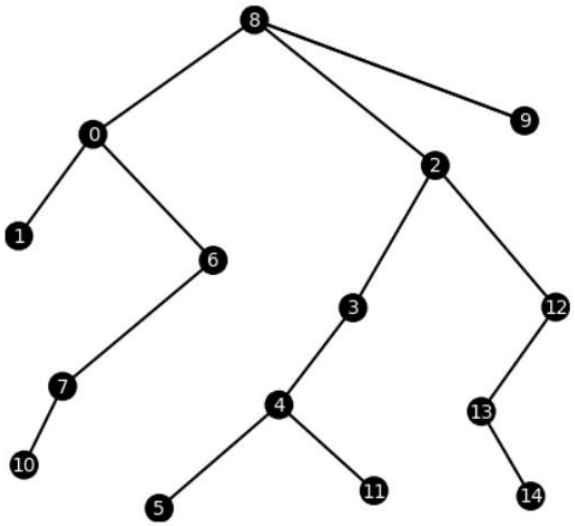


Bild 15: Ergebnis beim Plotten des konstruierten Graphen

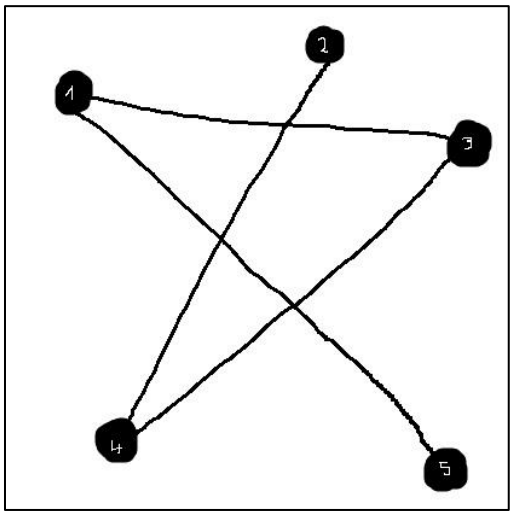


Bild 16: Eingabebild 4

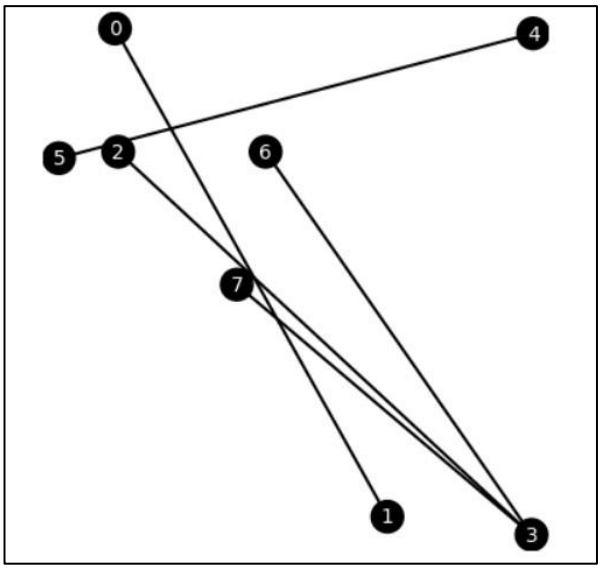


Bild 17: Ergebnis beim Plotten des konstruierten Graphen

9. Aufgetretene Probleme

Das Projekt weiß nicht, wie es weitergehen soll, ohne auf Schwierigkeiten oder weniger gute Dinge zu stoßen. Die größte Schwierigkeit bestand, wie oben erwähnt, bei der Bewertung des Modells, da das Modell leichte Mängel aufwies.

- Das erste Problem besteht in der exakten Erkennung von Kanten bei Kantenüberschneidungen. Das Modell hatte Schwierigkeiten, genau die gewünschten Boxen zu erkennen. Diese ungenaue Kantenerkennung führte dann wiederum auch zu einer schlechten Extraktion der Kantenenden und damit zu einem schlechten Clustering und man erhielt am Ende den falsch konstruierten Graphen.

- Anderes Problem war die Anpassung des Clustering Parameter, die aufwändig war: Beim DBSCAN-Clustering müssen wir der Parameter eps setzen. **eps** ist Maximale Distanz zwischen zwei Punkten, damit sie als Teil desselben Clusters (Knotens) betrachtet werden. Aber Wir wissen nicht den geschätzten Radius, die die Knoten besitzen könnten und darüber hinaus können die Knoten verschiedene Größe haben. Das heißt, mit einem ungeeigneten Parameter erhalten wir am Ende ein Graph, der nicht korrekt ist. Je nach vorherzusagendem Bild müssen wir den richtigen Wert von eps anpassen, damit das Clustering erfolgreich wird.

Quellen:

GitHub - python-pillow/Pillow: Python Imaging Library (Fork)

https://github.com/ultralytics/ultralytics

https://docs.ultralytics.com/quickstart/

https://github.com/adityaintwala/Hough-Line-Detection