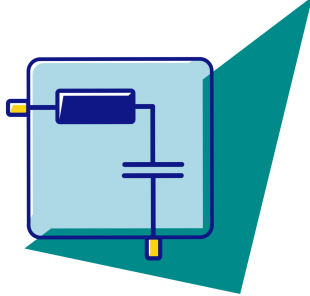


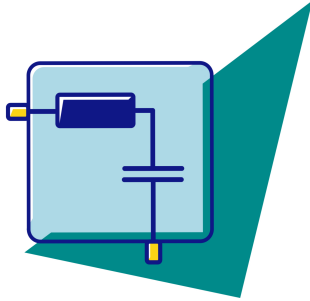
Smarter Circuits

Christoph Weberbauer, Markus Mayrhofer

Diplomarbeit - Dokumentation

Namen der Verfasserinnen / Verfasser	Christoph Weberbauer Markus Mayrhofer-Reinhartshuber
Jahrgang Schuljahr	5BHEL 2021/22
Thema der Diplomarbeit	Smarter Circuits - Entwicklung einer Handschrifterkennung für elektronische Schaltungen.
Kooperationspartner	keine
Aufgabenstellung	Die Simulation einer Hardware-Schaltung erfolgt meist in 2 Schritten. Zuerst wird die Schaltung manuell berechnet und am Papier skizziert. Danach wird sie in ein Simulationsprogramm übernommen und simuliert. Die Schaltung muss also mindestens zweimal gezeichnet werden, was einen erhöhten Arbeitsaufwand darstellt. Eine Software vereinfacht diesen Prozess durch die Umwandlung einer mit der Hand gezeichneten Schaltung in eine digitale Repräsentation der Schaltung.
Realisierung	Simulationen elektronischer Schaltungen sollen mit diesem Projekt komfortabel auf eine völlig neue Weise durchgeführt werden. Handschriftlich gezeichnete Schaltungen werden dabei erkannt und in ein, von bereits vorhandenen Simulatoren, verständliches Format umgewandelt.
Ergebnisse	Eine Software erlaubt es ein Bild der gezeichneten Schaltung zu laden. Anschließend müssen einige Einstellungen bezüglich der Umwandlung in ein Schwarz/ Weiß Bild getroffen werden. Wurden diese getätigt, wird die Schaltung analysiert und in dem Simulator LT-Spice geöffnet.
Typische Grafik, Foto etc.	
Teilnahme an Wettbewerben, Auszeichnungen	
Möglichkeiten der Einsichtnahme in die Arbeit	
	Logo keine
	Schulbibliothek der HTBLuVA Salzburg

Diploma thesis - Documentation

Author(s)	Christoph Weberbauer Markus Mayrhofer-Reinhartshuber
Form	5BHEL
Academic year	2021/22
Topic	Smarter Circuits – development of handwriting recognition for electronic circuits.
Co-operation Partners	none
Assignment	The simulation of a hardware circuit is usually done in 2 steps. First, the circuit is calculated manually and sketched on paper. Then it is transferred to a simulation program and simulated. Thus, the circuit must be drawn at least twice, which is an increased amount of work. A software simplifies this process by converting a hand-drawn circuit into a digital representation of the circuit.
Implementation	Simulations of electronic circuits are to be accomplished with this project, comfortably in a completely new way. Hand-drawn circuits are recognized and converted into a format that can be understood by existing simulators.
Results	A software allows to load an image of the drawn circuit. Then some settings must be made, regarding the conversion to a black/white image. Once these have been made, the circuit is analysed and opened in the LT-Spice simulator.
Illustrative Graph, Photo (incl. explanation) Participation in Competitions, Awards	
Accessibility of Diploma Thesis	Logo none School library of the HTBLuVA Salzburg

Inhaltsverzeichnis

1	Inhaltsverzeichnis	4
2	Einleitung	5
3	Individuelle Zielsetzung und Terminplan	5
3.1	Gantt-Diagramm - Markus Mayrhofer	6
3.2	Gantt-Diagramm - Christoph Weberbauer	6
4	GUI	7
4.1	Problem	7
4.2	Idee	7
4.3	Umsetzung	7
4.3.1	Bild öffnen	7
4.3.2	Bild zuschneiden	8
4.3.3	Binärbild erstellen	9
4.3.4	Bild in Schaltung umwandeln	10
5	Bildbearbeitung mit OpenCV	12
5.1	Problem	12
5.2	Idee	12
5.2.1	Problem bei Spulen	12
5.2.2	Lösung für Spulen	12
5.3	Umsetzung	13
5.3.1	Durchschnittliche Dicke einer Linie	13
5.3.2	Entfernen von Spulen	13
5.3.3	Umwandeln des Bildes	15
6	Entfernen von Mustern	16
6.1	Problem	16
6.2	Idee	16
6.3	Umsetzung	16
7	Graph Library	17
7.1	Begriffe	17
7.1.1	Graph	17
7.2	Problem	17
7.3	Idee	18
7.4	Umsetzung	19
7.4.1	iGraph (Bibliothek)	19
7.4.2	Selbst Programmierter Bibliothek	19
7.4.3	Weitere wichtige Algorithmen:	20
8	Umwandeln der Schaltung in eine Graph Datenstruktur	28
8.1	Begriffe	28
8.1.1	Richtungs-Gradient	28
8.2	Problem	28
8.3	Idee	28
8.4	Umsetzung	30
8.5	Algorithmus	31
9	Finden der Bauteile	34

9.1	Problem	34
9.2	Idee	34
9.3	Umsetzung	34
10	Bauteilklassifizierung	36
10.1	Problem	36
10.2	Idee	36
10.3	Neuronales Netzwerk	36
10.3.1	Aufbau	36
10.3.2	Trainieren eines Neuronalen Netzwerkes	38
10.4	Convolutional Neural Network	39
10.4.1	Filter	39
10.4.2	Beispiel Filter	39
10.4.3	Convolutional Layer	39
10.4.4	Pooling Layer	40
10.4.5	Beispiel Max pooling	40
10.4.6	Fully Connected Layer	40
10.5	Umsetzung	41
11	Rotationserkennung	42
11.1	Problem	42
11.2	Idee	42
11.3	Rotationserkennung mit Neuronalem Netzwerk	42
11.4	Umsetzung	42
12	Umstrukturieren des Graphen	43
12.1	Problem	43
12.2	Idee	43
12.3	Umsetzung	44
12.3.1	Konvertierung zu relativen Koordinaten	44
12.3.2	Umstrukturieren des Graphen	44
12.3.3	Bauteile gerade richten	46
12.3.4	Beschreibung des Algorithmus	46
12.3.5	Bauteile gerade richten (nicht genutzt)	47
13	Output File erstellen	48
13.1	Problem	48
13.2	Idee	48
13.3	LT-Spice Syntax	48
13.4	Umsetzung	49
14	Abbildungsverzeichnis	53

2 Einleitung

Smarter Circuits ist eine Applikation für Techniker und Technikerinnen. Die Simulation einer Hardware-Schaltung erfolgt meist in 2 Schritten. Zuerst wird die Schaltung manuell berechnet und am Papier skizziert. Danach wird sie in ein Simulationsprogramm übernommen und simuliert. Die Schaltung muss also mindestens zweimal gezeichnet werden, was einen erhöhten Arbeitsaufwand darstellt. Die Lösung dafür ist eine Software, welche die mit der Hand gezeichneten Schaltungen analysiert und direkt in einem Simulatoren-Programm importiert.

Das Programm besteht hierbei lediglich aus einer einfach zu bedienenden grafischen Oberfläche, mit der man das Ausgangsbild auswählen kann. Mit dem Drücken eines Knopfes kann aus dem Bild dann eine Schaltung generiert werden, welche sogleich in LT-Spice geöffnet wird.

3 Individuelle Zielsetzung und Terminplan

Unser Projekt kann grob in folgendes Blockschaltbild unterteilt werden.

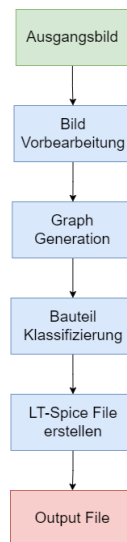


Abbildung 1: Blockschaltbild für unser Projekt

Aus diesem Blockschaltbild hat sich folgende Aufgabenteilung sinnvollerweise ergeben:

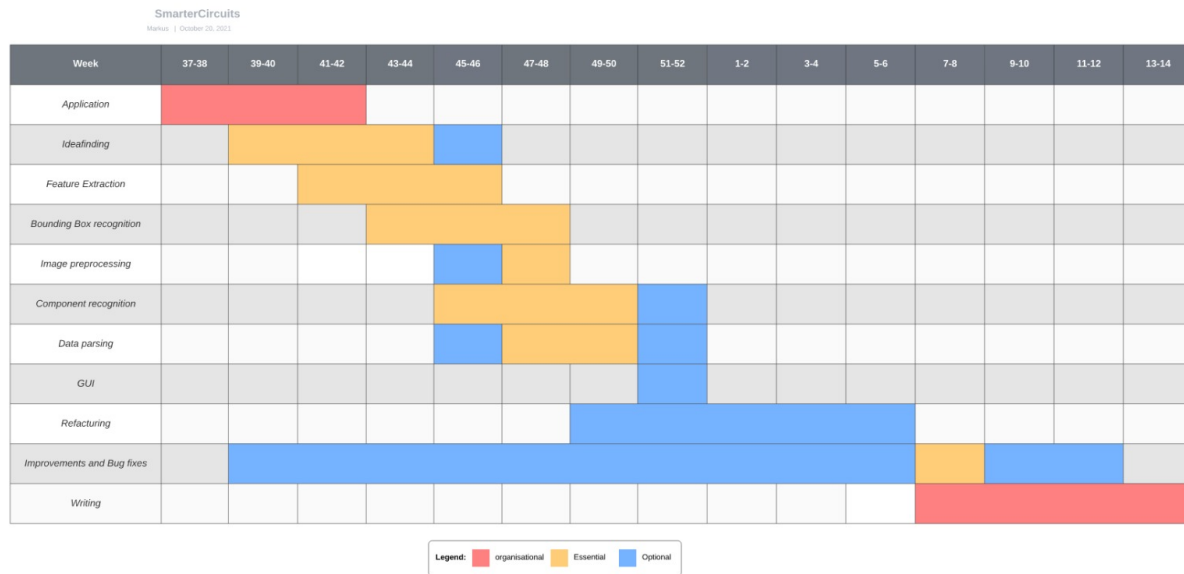
Markus Mayrhofer:

- Bild Vorbearbeitung
- Bauteil Klassifizierung

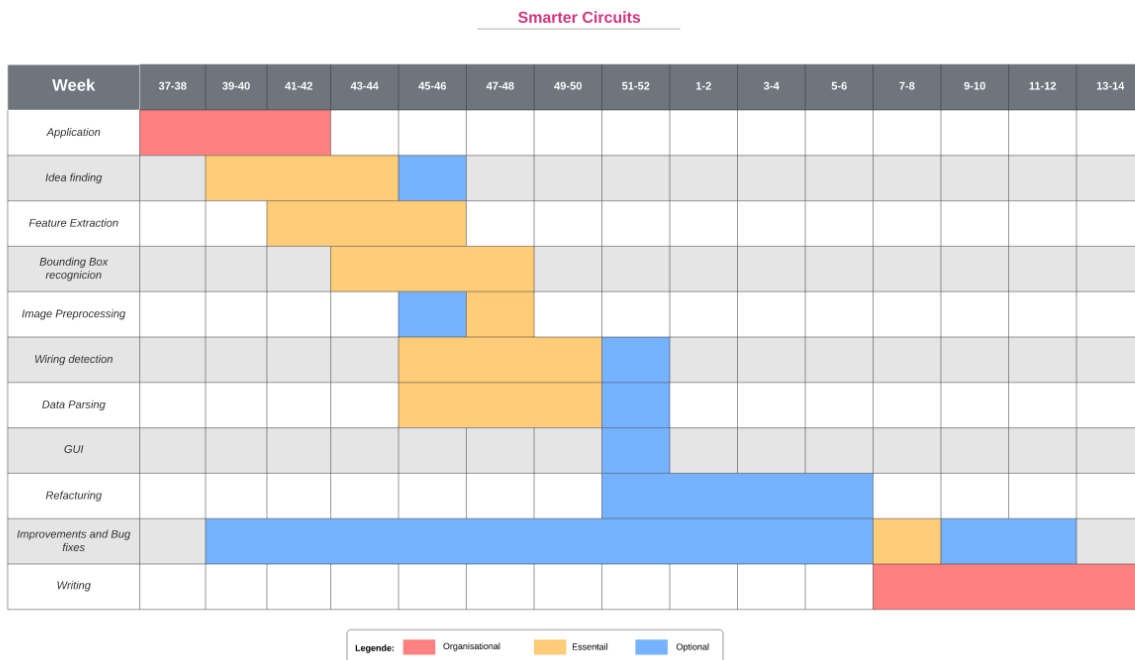
Christoph Weberbauer:

- Graph Generation
- LT-Spice File erstellen

3.1 Gantt-Diagramm - Markus Mayrhofer



3.2 Gantt-Diagramm - Christoph Weberbauer



4 GUI

4.1 Problem

Um das Bild in eine Schaltung umwandeln zu können, muss dieses erst in ein Binärbild umgewandelt werden. Dafür wird ein Algorithmus aus der Python Library OpenCV verwendet. Da jedoch jedes Bild unterschiedlich beleuchtet ist, ist es nicht möglich bei allen Bildern die selben Einstellungen zu benutzen.

4.2 Idee

Bei jedem Bild kann der Benutzer selbst gewisse Werte für den Algorithmus verändern und sieht das jeweilige Ergebnis. So kann jedes Bild in ein optimales Binärbild umgewandelt werden.

4.3 Umsetzung

Die GUI ist in verschiedene Fenster unterteilt. Jedes dieser Fenster ist für einen Teil zuständig. Mit den Knöpfen "Next" und "Back" kann jederzeit ein Fenster nach vorne oder zurück gewechselt werden.

4.3.1 Bild öffnen

Wird das Programm gestartet, muss man zuerst ein Bild öffnen.

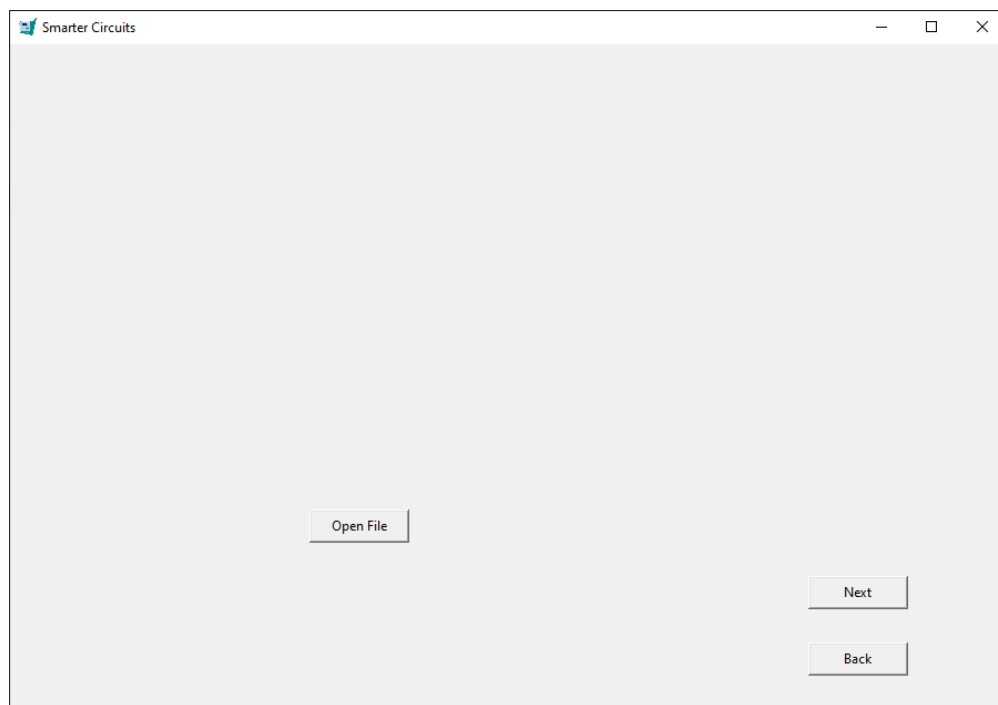


Abbildung 2: Bild öffnen

Wird der Knopf "Open File" gedrückt, öffnet sich in einem neuen Fenster der File Explorer. In diesem kann der PC nach einem Bild durchsucht werden. Mit dem Knopf "Open" wird das gewünschte Bild ausgewählt.

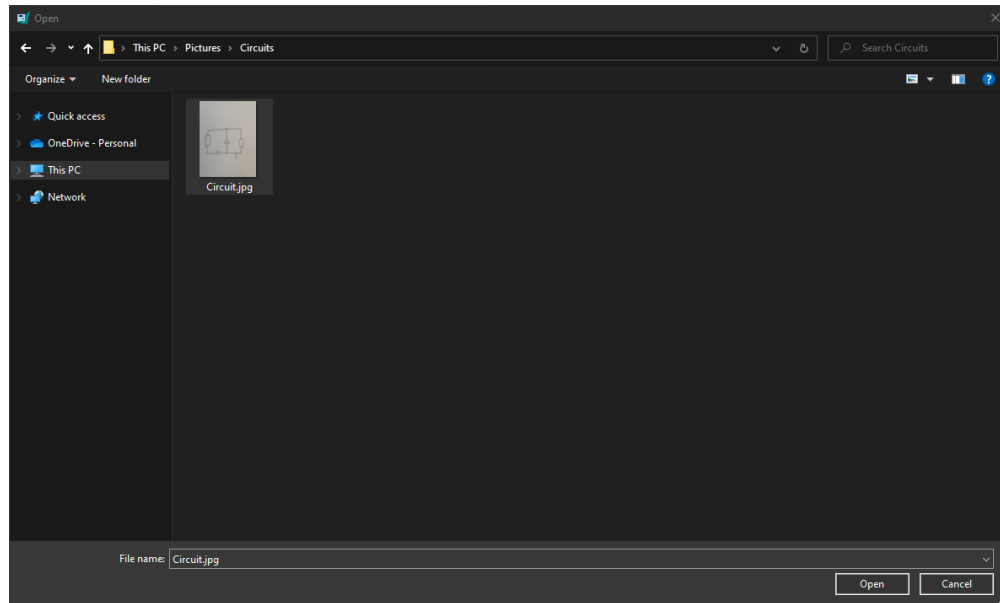


Abbildung 3: Bild im File Explorer auswählen

4.3.2 Bild zuschneiden

Das Ausgewählte Bild wird jetzt im Program angezeigt und kann noch zugeschnitten werden, damit auf dem Bild nur noch die Schaltung zu sehen ist. Dafür wird mit der Maus auf einen Punkt geklickt und mit gedrückter Maustaste die Maus zu einem zweiten Punkt bewegt. Das dadurch entstehende Rechteck zeigt den Teil des Bildes, welcher ausgeschnitten wird. Der restliche Teil des Bildes wird ausgegraut dargestellt.

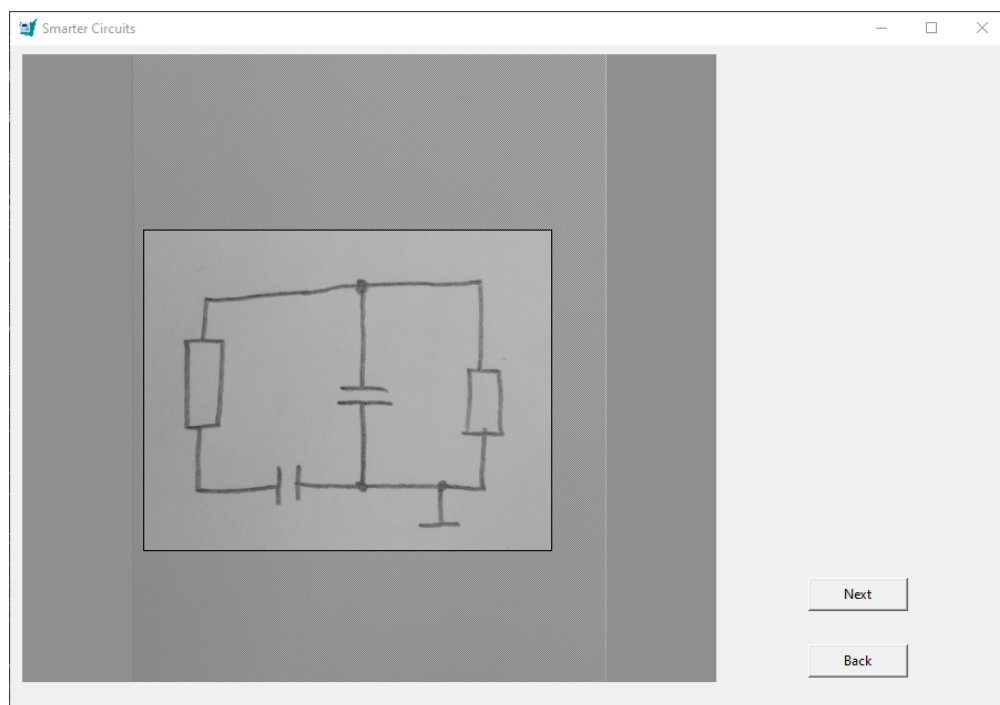


Abbildung 4: Bild zuschneiden

4.3.3 Binärbild erstellen

Als nächstes wird das Bild in ein Binärbild umgewandelt.

Dazu wird eine der zwei Funktionen "threshold" oder "adaptiveThreshold" aus der Library OpenCV verwendet.

Der Funktion "threshold" wird ein Schwellenwert zwischen 0 und 255 übergeben. Dieser Wert kann mithilfe des zweiten Schiebereglers eingestellt werden. Jeder Pixel mit einem Wert unter dem Schwellenwert wird auf 0 gesetzt, jeder mit einem höheren Wert auf 255. Um ein besseres Ergebnis zu erzielen, kann mit dem ersten Schieberegler zusätzlich die Funktion "GaussianBlur" das Bild unscharf machen, um Rauschen aus dem Bild zu entfernen.

Diese funktioniert jedoch nur bei gleichmäßig ausgeleuchteten Bildern.

Das Ergebnis dieser Funktion bei ungleichmäßig ausgeleuchteten Bildern ist in der unteren Abbildung zu sehen. Für diese Fälle gibt es eine zweite Möglichkeit ein Bild in ein Binärbild umzuwandeln.

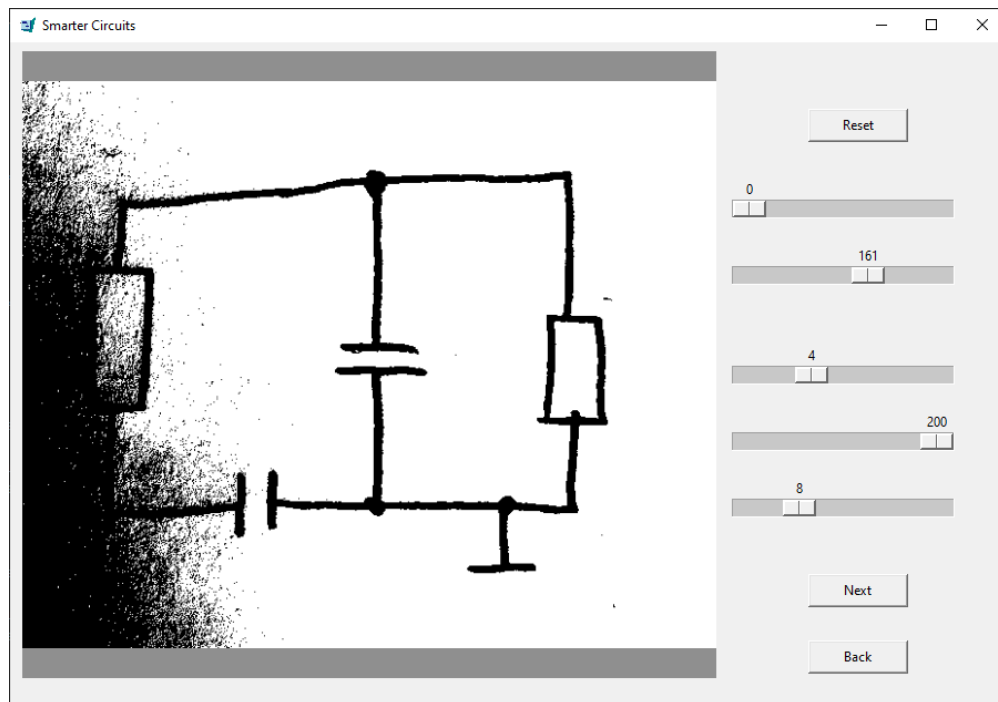


Abbildung 5: Binärbild erstellen

Die zweite Funktion "adaptiveThreshold" benutzt für jede Bildregion einen anderen Schwellenwert der von der Beleuchtung in dem jeweiligen Bereich abhängt. Dadurch ist es auch möglich bei ungleichmäßig ausgeleuchteten Bildern ein Binärbild zu erstellen. Mit dem vierten und fünften Schieberegler können die beiden Parameter für diese Funktion verändert werden. Wie bei der ersten Methode kann auch wieder hier mit dem dritten Schieberegler das Bild unscharf gemacht werden, um ein besseres Ergebnis zu erzielen.

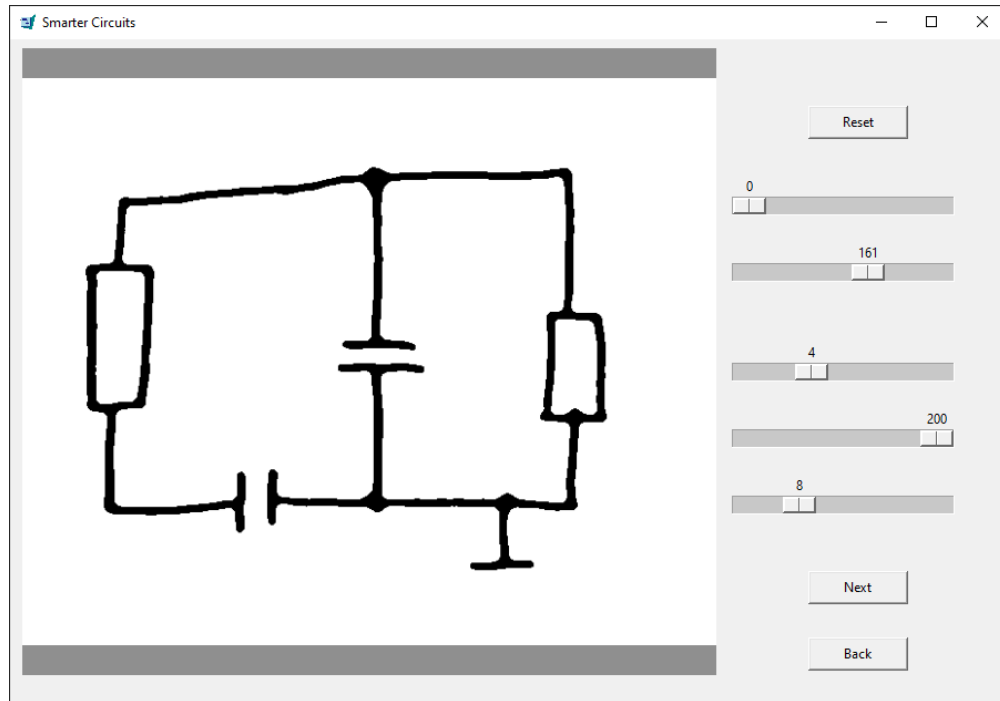


Abbildung 6: Binärbild erstellen

Mit dem Knopf "Reset" kann das Bild wieder auf das Ausgangsbild zurückgesetzt werden.

4.3.4 Bild in Schaltung umwandeln

Zuletzt kann die Schaltung mit dem Knopf "Detect Circuit" in ein LTspice File umgewandelt werden.

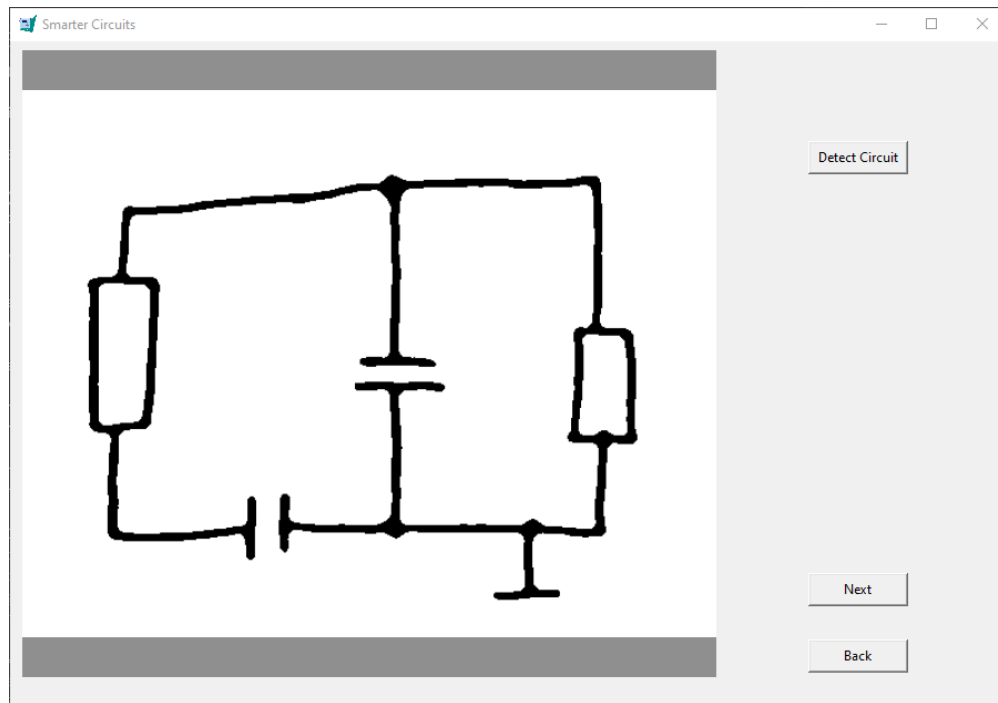


Abbildung 7: Bild umwandeln

Wenn die Schaltung fertig umgewandelt wurde, wird LTSpice gestartet und die erstellte Datei wird geöffnet.

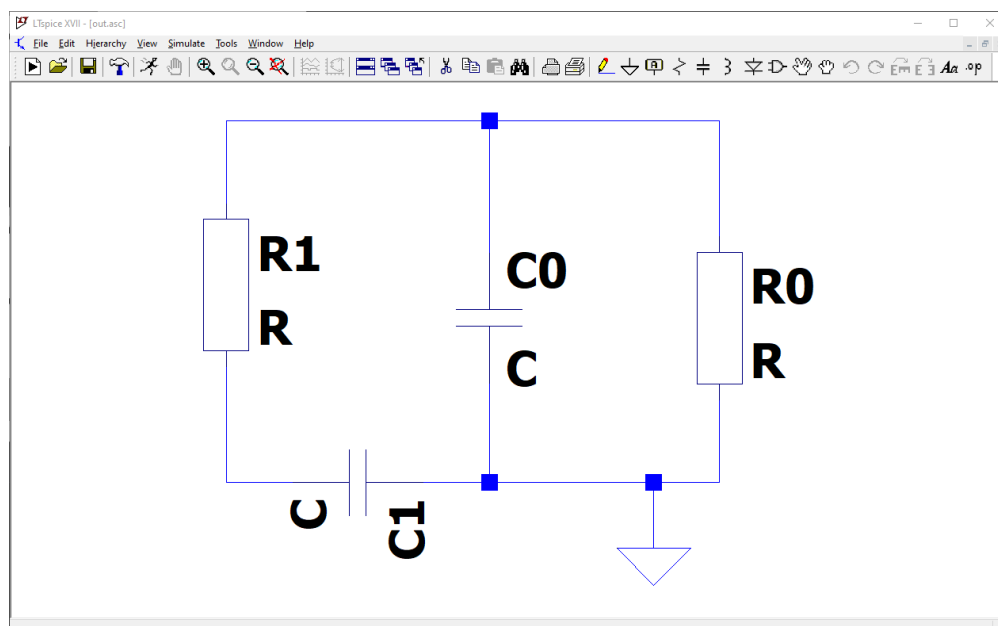


Abbildung 8: Schaltung in LTSpice

5 Bildbearbeitung mit OpenCV

5.1 Problem

Um das Bild der Schaltung weiter verarbeiten zu können, ist es wichtig, alle Linien in dem Bild der Schaltung auf eine Breite von einem Pixel zu verändern.

5.2 Idee

Mithilfe einer Funktion aus der Python Library OpenCV soll jede Linie in dem Bild in eine Linie mit einer Breite von einem Pixel umgewandelt werden.

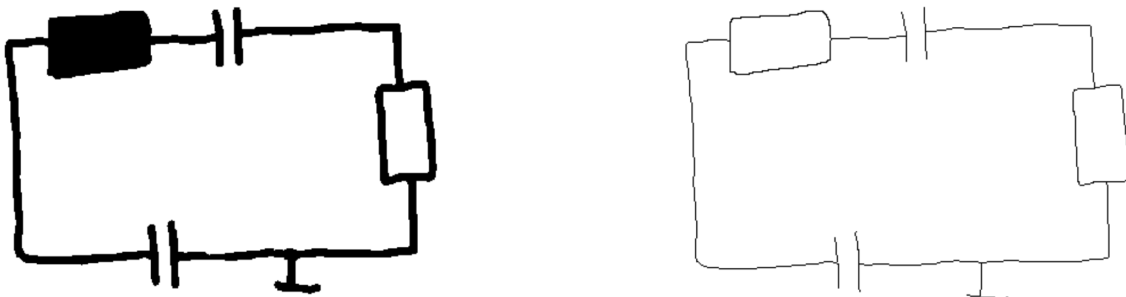


Abbildung 9: Binärbild und gewünschtes Ergebnis

5.2.1 Problem bei Spulen

Dieser Algorithmus funktioniert jedoch bei Spulen nicht, da diese für den Algorithmus nur eine breitere Linie, wie die Verbindungen zwischen den Bauteilen darstellen und somit auch zu einer ein Pixel breiten Linie geändert werden.

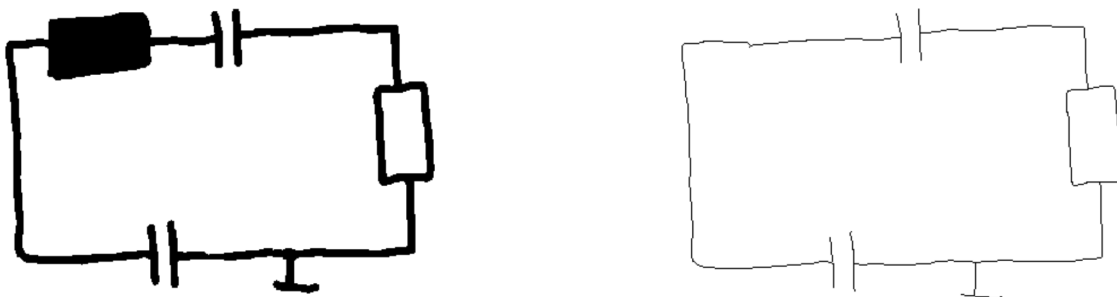


Abbildung 10: Problem bei Spulen

5.2.2 Lösung für Spulen

Um zu verhindern dass Spulen entfernt werden, werden zuerst die schwarzen Flächen in den Spulen entfernt. Spulen haben somit die gleiche Struktur wie Widerstände und können in eine ein Pixel breite Linie umgewandelt

werden.

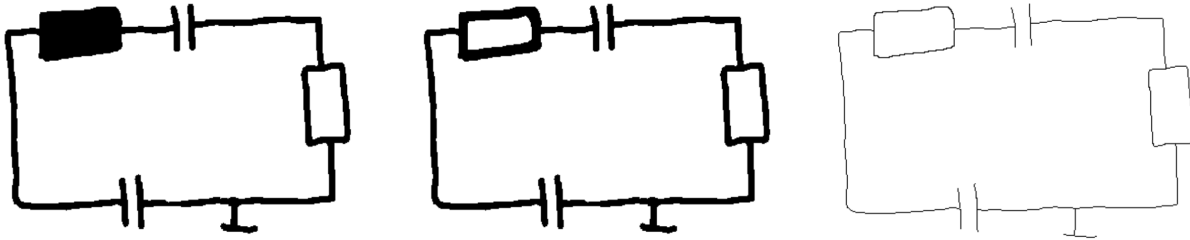


Abbildung 11: Lösung für Spulen

5.3 Umsetzung

5.3.1 Durchschnittliche Dicke einer Linie

Um Spulen entfernen zu können, muss zuerst die durchschnittliche Dicke der Linien in der Schaltung herausgefunden werden.

Dafür wird in jeder Spalte jeder Pixel nacheinander überprüft. Sobald ein Pixel schwarz ist, hat man eine Linie gefunden. Ein Zähler wird ab diesem Pixel so lange erhöht bis wieder ein weißer Pixel auftritt und die gezeichnete Linie somit endet. Nach jeder Linie wird die Dicke der Linie in dieser Spalte in eine Liste gespeichert.

Beispiel einer Linie welche in der rot markierten Spalte drei Pixel breit ist.

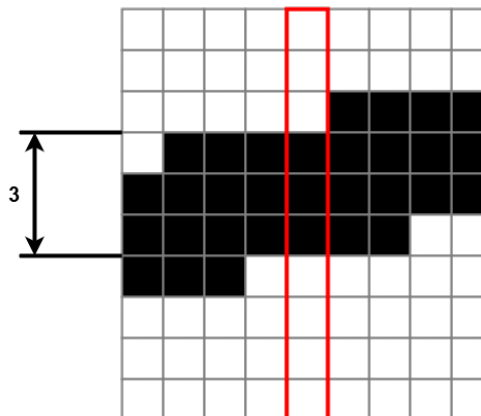


Abbildung 12: Linien Dicke in einer Spalte

Als durchschnittliche Liniendicke wird der Wert gewählt, welcher bei den in der Liste gespeicherten Breiten am häufigsten auftritt.

5.3.2 Entfernen von Spulen

Um Spulen zu entfernen wird die Funktion "erode" aus der Library OpenCV verwendet. Dabei wird eine Filtermaske über ein Bild geschoben. Ein Pixel im Originalbild bleibt nur dann weiß, wenn alle Pixel unter dem Filter weiß sind.

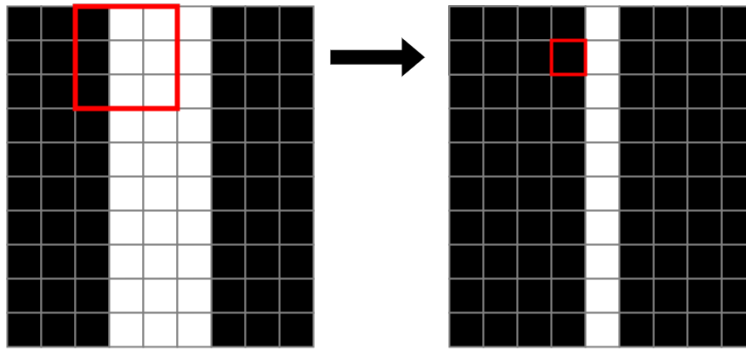


Abbildung 13: Funktion "erode" Beispiel 1

Unter dem 3x3 Filter sind schwarze und weiße Pixel somit wird der Pixel im Zentrum des Filters schwarz.

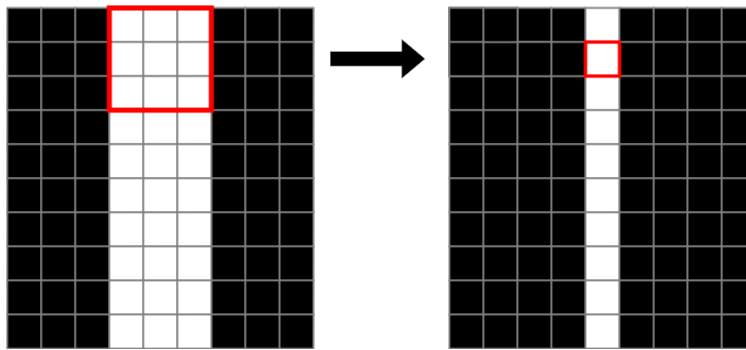


Abbildung 14: Funktion "erode" Beispiel 2

Wird der Filter einen Pixel weiter nach rechts bewegt, sind alle Pixel unter dem Filter weiß, der Pixel bleibt somit weiß.

Damit ist es möglich dicke Linien dünner zu machen, sowie dünne Linien zu entfernen.

Wird nun das Binäre Bild invertiert, und diese Funktion mit einer Filtergröße von der dreifachen zuvor ausgerechneten Liniendicke angewandt, bleibt nur noch die Spule zurück.

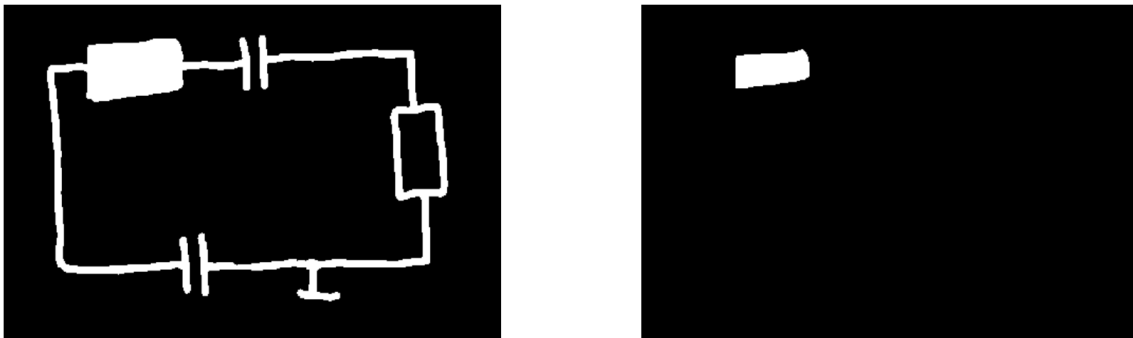


Abbildung 15: Entfernen der Schaltung bis auf Spulen

Danach wird die Differenz zwischen dem Invertiertend Binärbild und dem "Erosion" Bild gebildet. Dadurch wird

die Fläche in der Spule entfernt. Zuletzt wird das Bild erneut invertiert, um wieder auf das ursprüngliche Format zurückzukommen.

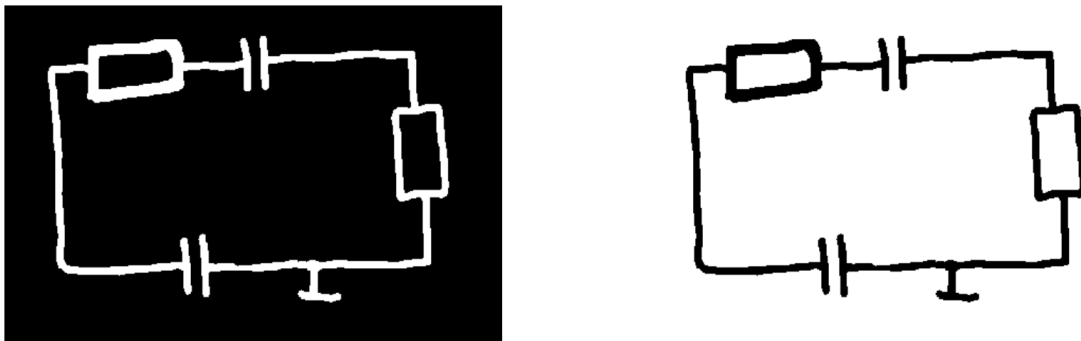


Abbildung 16: Differenz zwischen den beiden Bildern

5.3.3 Umwandeln des Bildes

Um alle Linien in eine Breite von einem Pixel umzuwandeln wird die Funktion "thinImage" aus der Library OpenCV verwendet. Diese liefert als Rückgabewert das konvertierte Bild.

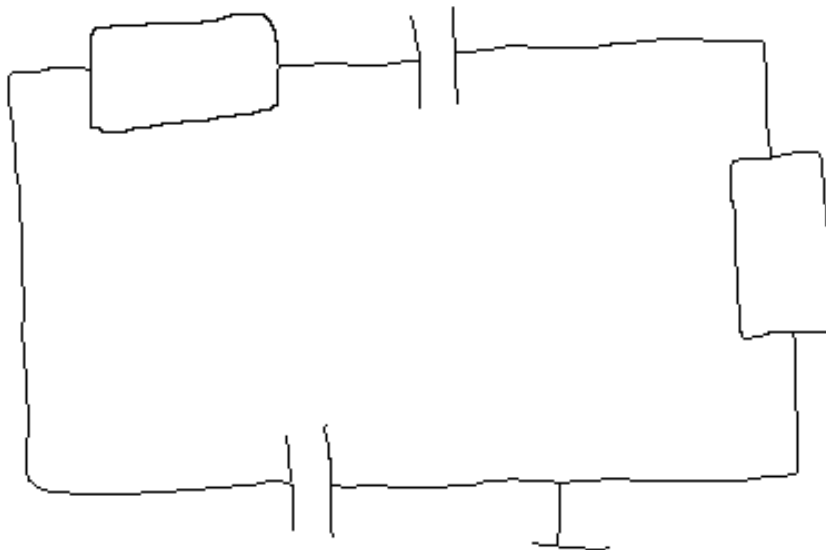


Abbildung 17: Bild mit einer Linienbreite von einem Pixel

6 Entfernen von Mustern

6.1 Problem

Damit das Bild in eine Schaltung umgewandelt werden kann, darf es bestimmte Muster in dem Bild der Schaltung nicht geben, da es bei diesen bei der weiteren Verarbeitung der Schaltung zu Fehlern kommen würde.

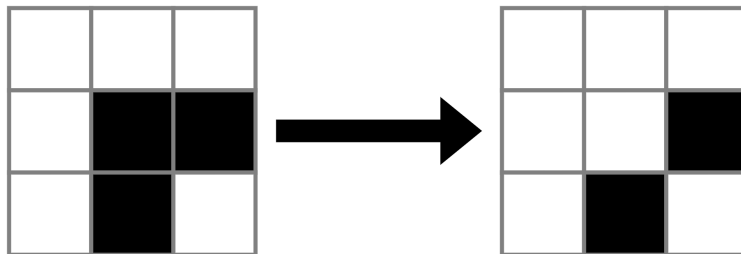


Abbildung 18: Beispiel eines zu Entfernendes Muster

6.2 Idee

In dem Bild der Schaltung werden diese bestimmten Muster gesucht und so verändert, dass es später nicht mehr zu Fehlern kommen kann.

6.3 Umsetzung

Jedes Muster welches erstellt werden soll wird in einer Liste gespeichert. In einer weiteren Liste wird das jeweilige Muster gespeichert welches das erste ersetzen soll.

Danach wird jeder schwarze Pixel im Bild überprüft, ob dieser Pixel sowie die angrenzenden Pixel gleich wie ein zu ersetzendes Muster gefunden, werden die entsprechenden Pixel im ursprünglichen Bild mit dem zuvor festgelegtem Ersatzmuster ausgetauscht.

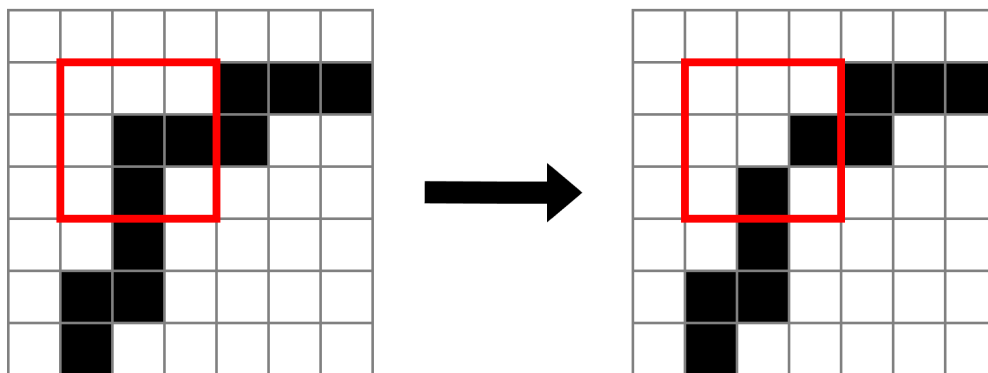


Abbildung 19: Entfernen eines Musters

7 Graph Library

7.1 Begriffe

7.1.1 Graph

Ein Graph ist eine Sonderform einer Baum-Datenstruktur.

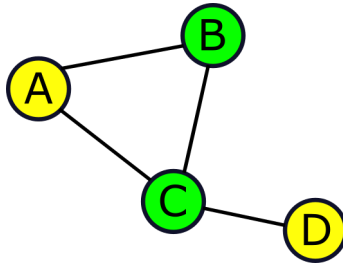


Abbildung 20: Beispiel für einen Graphen

Er besteht aus sogenannten "Vertices" (Singular Vertex) und "Edges".



Abbildung 21: Bestandteile eines Graphen

Wobei je ein Vertex einen Datenpunkt repräsentiert, welcher mithilfe einer Edge mit anderen Datenpunkten verbunden ist. Jeder Vertex und jede Edge kann auch noch eine bestimmte Farbe haben. So kann zwischen zwei Strukturen, die zwar strukturell gleich sind, aber je andere Dinge darstellen, unterschieden werden.

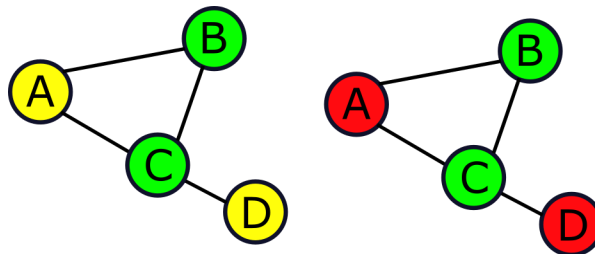


Abbildung 22: Strukturell gleiche Graphen, welche sich lediglich durch die Farbe unterscheiden

In jedem Vertex können hierbei auch unterschiedliche Daten, wie zum Beispiel: Koordinaten gespeichert werden, diese unterscheiden zwei Graphen allerdings **nicht** voneinander!

7.2 Problem

Die gesamte elektronische Schaltung soll später mithilfe eines Graphen dargestellt werden. Dafür ist es nötig, eine Möglichkeit zu schaffen, sehr einfach Graphen zu erstellen und diese weiter zu verarbeiten.

7.3 Idee

Eine einfache Bibliothek finden/erstellen. Diese Bibliothek muss mindestens folgende Funktionen bieten:

- Graphen erstellen
- Graphen zeichnen
- Einzelne Vertices erstellen und Daten in diese speichern
- Vertices löschen
- Vertices mit Edges verbinden
- Edges löschen
- Nachbar-Vertices herausfinden (siehe Abbildung:Nachbar-Vertices23)
- Vertices gruppieren und durch einen einzelnen Vertex ersetzen (siehe Abbildung:Vertices-Gruppieren24)
- Einen Vertex zwischen zwei anderen einfügen (siehe Abbildung:Zwischen-Vertices25)
- Zwei Graphen zusammenfügen
- Muster in Graphen zu finden (siehe Abbildung:Muster-Finden26)

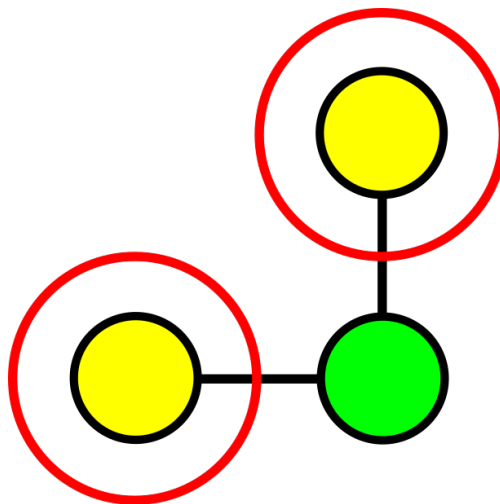


Abbildung 23: Nachbar-Vertices

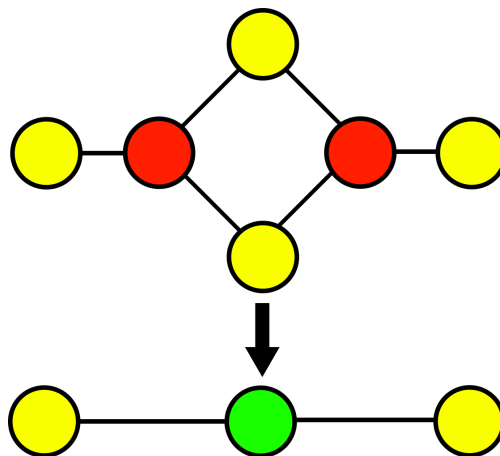


Abbildung 24: Vertices-Gruppieren

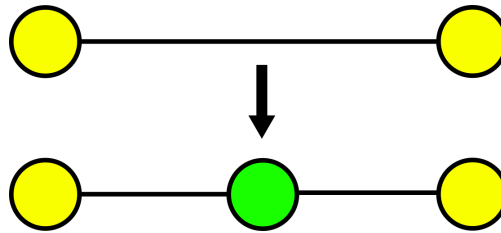


Abbildung 25: Zwischen-Vertices

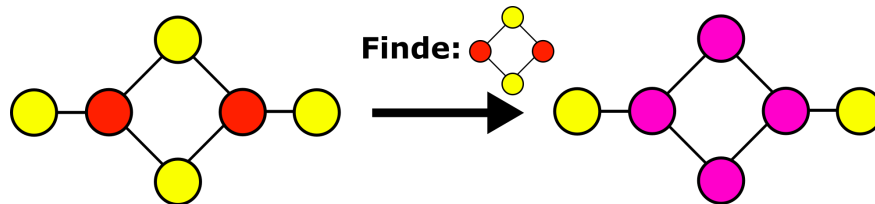


Abbildung 26: Muster-Finden

7.4 Umsetzung

7.4.1 iGraph (Bibliothek)

Zuerst wurde die Bibliothek iGraph verwendet, um mit Graphen zu arbeiten. Ausgewählt wurde dies aufgrund der Verfügbarkeit der Funktion `get_isomorphisms_vf2()`. Diese kann schnell und zuverlässig Muster in Graphen finden. (siehe Erklärung zuvor)

Problem:

- 1) Die Dokumentation der Bibliothek ist sehr schlecht.
- 2) Außerdem führt ein Löschen von Vertices zu großen Problemen.
- 3) Dadurch sind Funktionen wie "gruppieren", "einfügen von Vertices" nicht umsetzbar.

7.4.2 Selbst Programmierte Bibliothek

Lösung:

Selbst eine Bibliothek entwickeln, welche die Probleme von iGraph behebt. Da die Bibliothek selber in Python entwickelt wurde, ist sie allerdings relativ langsam, was in diesem Fall jedoch nicht sonderlich stört. IGraph wird dann lediglich während der Entwicklung der Software zum Debuggen (zeichnen von Graphen) verwendet.

Implementation:

Im Hintergrund wird ein Graph mithilfe einer Inzidenzmatrix gespeichert. Eine Inzidenzmatrix ist hierbei eine Tabelle, deren Reihen je einen Vertex und deren Spalten je eine Edge beschreiben. Eine 1 bei der Edge y und bei dem Vertex x heißt dann einfach, der Vertex x ist mit der Edge y verbunden.

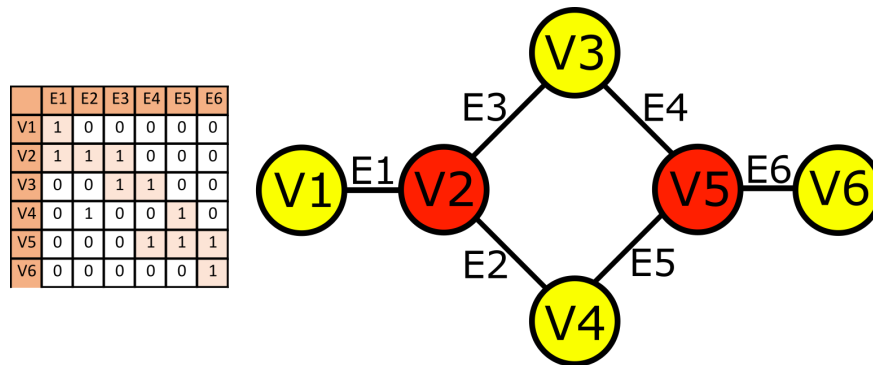


Abbildung 27: Beispiel für eine Inzidenzmatrix

In Beispiel 27 sieht man: Vertex "1" ist mit der Edge "1" verbunden. Vertex "2" ist mit Edge "1","2","3" verbunden.

7.4.3 Weitere wichtige Algorithmen:

7.4.3.1 Zeichnen: Zum Zeichnen der Graphen wird ein Graph aus unserer Bibliothek zuerst in ein xml-encodiertes Text-Dokument umgewandelt. Dieses Dokument wird dann mithilfe von iGraph geöffnet und mit den dadurch zur Verfügung gestellten Funktionen gezeichnet. Dies wurde gemacht, da es wesentlich einfacher ist, als einen Graph-Zeichen-Algorithmus zu implementieren.

7.4.3.2 Muster finden: Verwendet wurde der sogenannte "Ulman's Subgraph Isomorphism Algorithm". Dieser basiert darauf, dass alle Möglichkeiten durchprobiert werden, bis alle Vorkommen eines Musters in einem Ausgangsgraphen gefunden wurden.

Eine wichtige Rolle spielen hierbei sogenannte "Zuordnungen". Eine Zuordnung ist dabei einfach eine Matrix, welche alle Vertices aus einem Graphen je einen anderen Vertex aus einem anderen Graphen zuordnet.

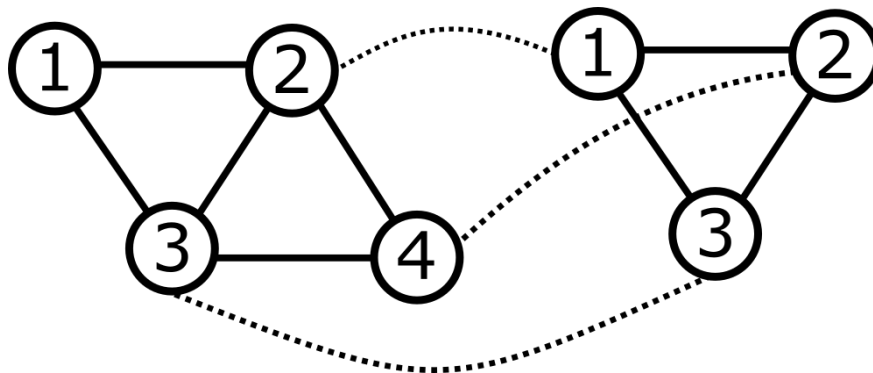


Abbildung 28: Beispiel für eine von mehreren möglichen Zuordnungen. Die jeweiligen Zuordnungen sind gepunktet dargestellt

Beschreibung des Grund-Algorithmus:

Sei H ein Graph und N das Muster, welches in H gefunden werden soll. H besteht aus $|V_H|$ Vertices und N aus $|V_N|$ Vertices

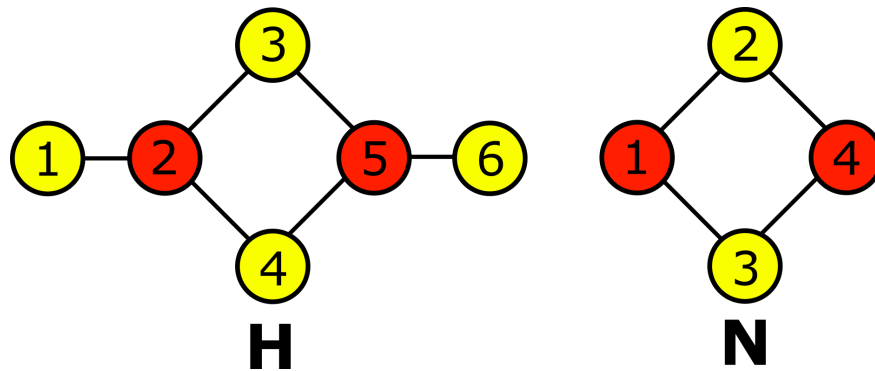


Abbildung 29: Beispiel für 2 Graphen

1) Erstelle eine $|V_H| \times |V_N|$ Matrix und befülle sie mit 0.

		H-Graph					
		1	2	3	4	5	6
N-Graph	1	0	0	0	0	0	0
	2	0	0	0	0	0	0
	3	0	0	0	0	0	0
	4	0	0	0	0	0	0

Abbildung 30: Leere Matrix für die oben gezeigten Beispiele

In dieser Matrix bezeichnet eine 1 an der Stelle x,y dann dass: der Vertex x in H dem Vertex y in N zugeordnet wird

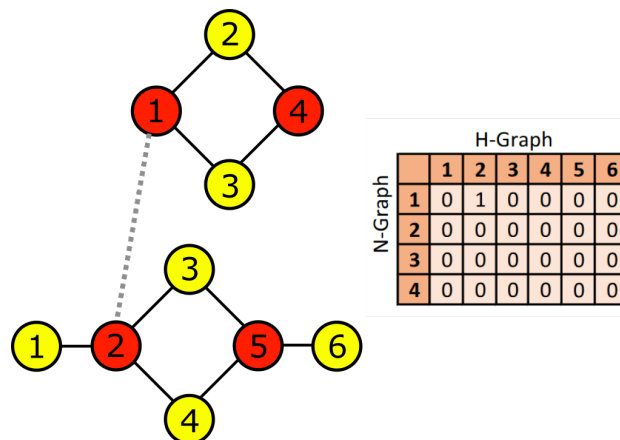


Abbildung 31: In diesem Fall ist Vertex 1 in N, Vertex 2 in H zugeordnet

2) Befülle die Matrix an allen Stellen mit 1, an denen eine Zuordnung möglich ist. Das bedeutet (x,y) ist 1 wenn Vertex x in H einem Vertex y in N zugeordnet werden kann! Für die Überprüfung siehe Algorithmus: 7.4.3.2

3) Starte in der ersten Reihe der Matrix und mit noch keiner besuchten Spalte.

4) Prüfe ob in der gesamten Tabelle pro Spalte **maximal** eine 1 steht und pro Zeile **genau** eine 1 steht. Das heißt jeder Vertex in N ist genau einem Vertex in H zugeordnet.

Wenn Ja) >Wenn die generierte Zuordnung auch eine mögliche ist, speichere sie. Für Algorithmus siehe 7.4.3.2
Wenn Nein)

5) Erstelle eine Kopie der Matrix

6) Für jede noch nicht besuchte Spalte, besuche Spalte:

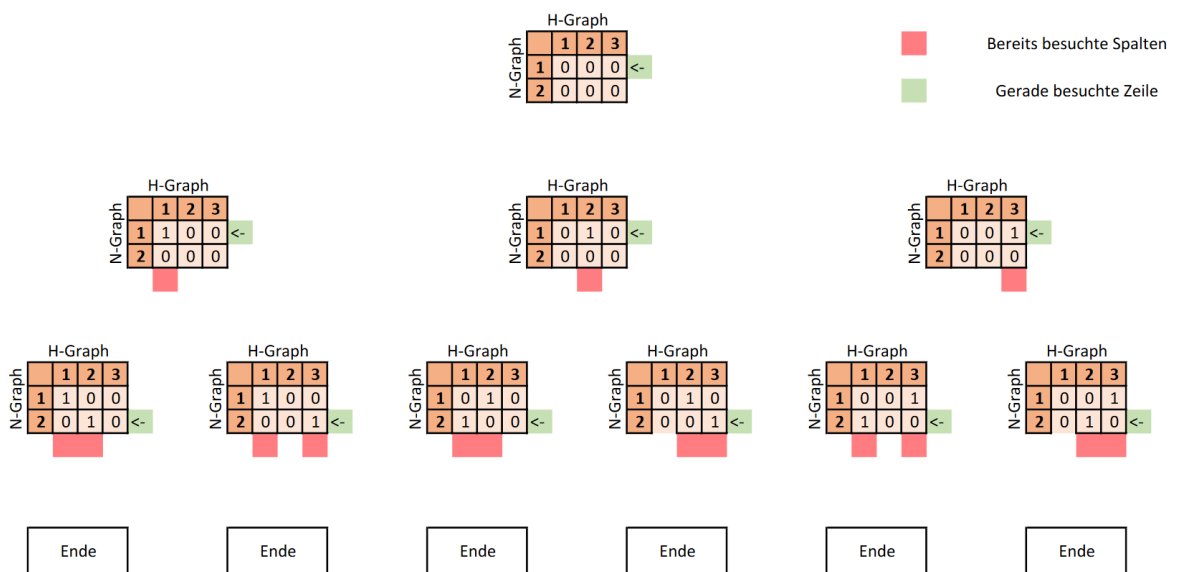
6.1) Prüfe ob der Wert in der Matrix an der derzeitigen Reihe und gerade besuchten Spalte 1 ist. Wenn nein, überspringe die Spalte

6.2) Sei die derzeit besuchte Spalte x und die derzeit besuchte Reihe y, setze (x,y) zu 1

6.3) Wiederhole ab Schritt 3) mit:

- Reihe+1
- Besuchte Spalten + gerade besuchter Spalte
- Kopie der Tabelle

6.4) Setze die Tabelle wieder zurück



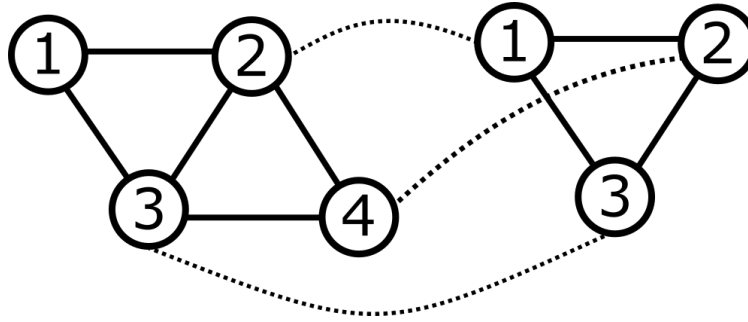


Abbildung 33: Beispiel für Überprüfung. Die Zuordnung ist dabei gepunktet

Für jeden Vertex in N, suche den zugeordneten Vertex aus H.

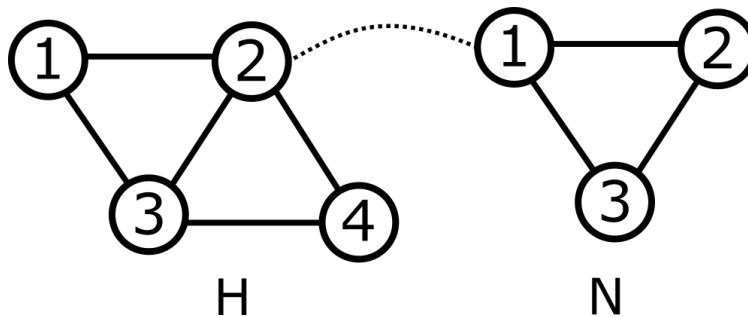


Abbildung 34: Suche den zugeordneten Vertex für 1

Suche die Nachbarn für beide Vertices.

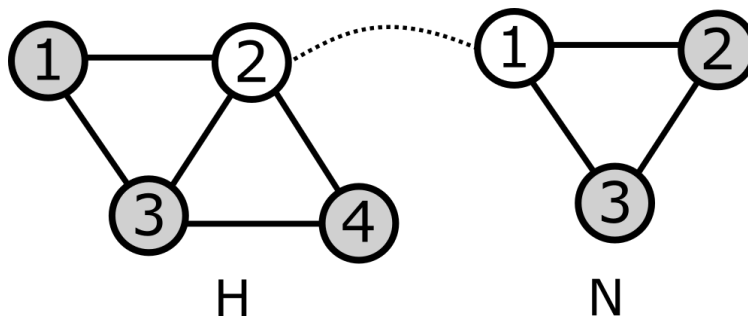


Abbildung 35: Suche die Nachbarn (hier grau)

Für jeden Nachbar des N-Vertex, suche dessen zugeordneten H-Vertex. Nur wenn mindestens einer davon auch ein Nachbar des ursprünglichen Vertex aus H ist, ist eine Zuordnung möglich

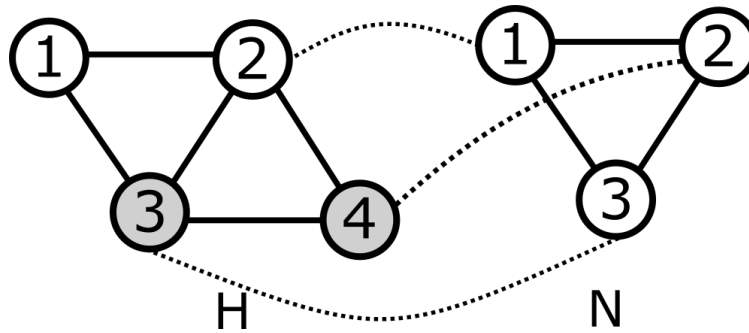


Abbildung 36: Zuordnung von 1 auf 2 ist in diesem Fall somit möglich!

Dies wird solange wiederholt, bis die Matrix nicht mehr verändert wurde.

Beschreibung des Algorithmus zur Prüfung ob eine Zuordnung möglich ist

Suche für jeden Vertex aus N, den zugeordneten Vertex aus H. Wenn beide nicht die gleiche Farbe haben -> keine Zuordnung möglich

Suche für jeden Vertex aus N, den zugeordneten Vertex aus H. Wenn der Vertex aus H weniger Nachbarn hat als der aus N -> keine Zuordnung möglich!

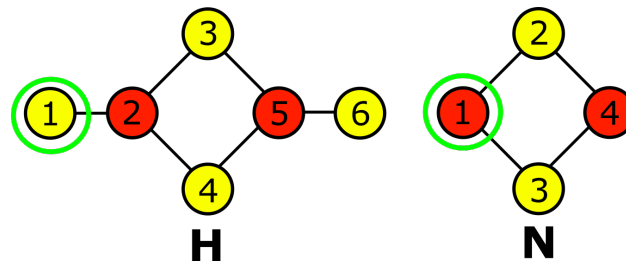


Abbildung 37: 1 und 1 werden nie zuordenbar sein, da sie unterschiedliche Farbe haben und H1 nur einen Nachbar hat, N1 aber mindestens zwei benötigt

Beischreibung der Überprüfung ob es sich um eine mögliche Zuordnung handelt

Die Idee ist folgende: Es wird aus einer Zuordnung und aus dem Graphen H ein neuer Graph generiert. Ist dieser Graph gleich dem Graphen N, so handelt es sich um eine valide Zuordnung!

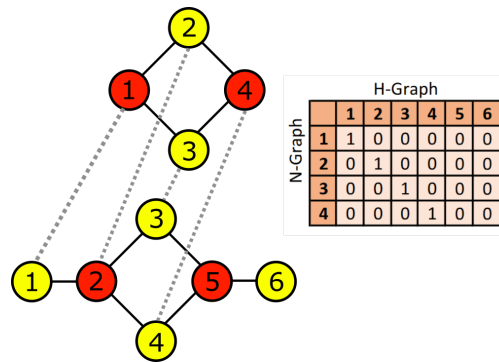


Abbildung 38: Beispiel für eine unmögliche Zuordnung

Dafür werden die Graphen H und N in eine sogenannte Adjacency-Matrix umgewandelt. Anschließend rechnet man:

$$M(MH)^T == N$$

wobei M die Zuordnungsmatrix ist

Graphisch repräsentiert diese Rechnung folgendes:

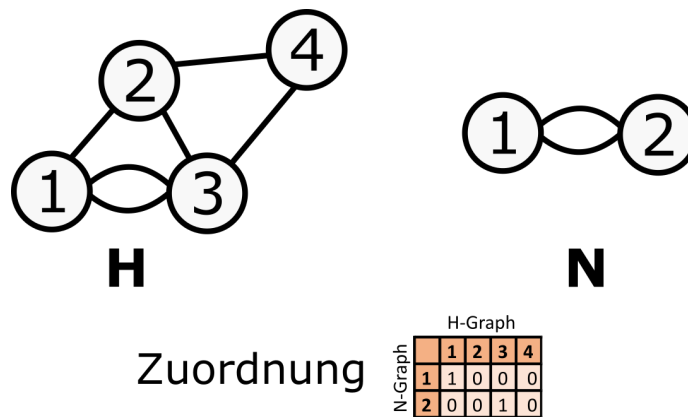


Abbildung 39: Beispiel Graphen

Schritt 1

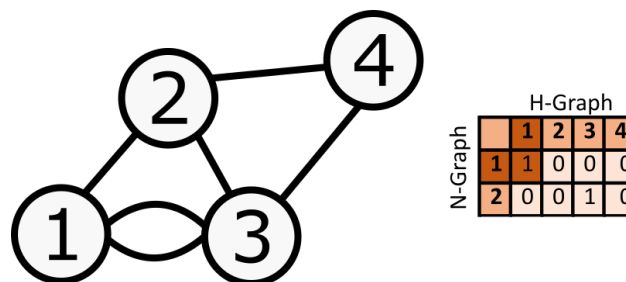


Abbildung 40: Alle Verbindungen nach 1 werden ausgetauscht mit Verbindungen nach Zuweisung(1) = 1

Schritt 2

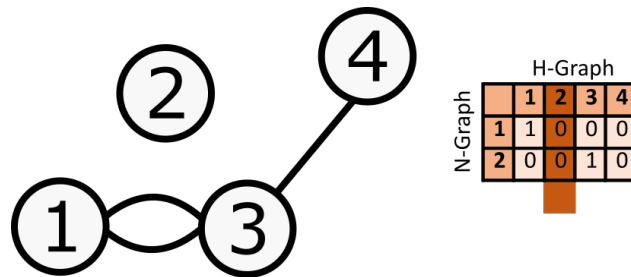


Abbildung 41: Alle Verbindungen nach 2 werden mit Verbindungen nach Zuweisung(2) = Keine

Schritt 3

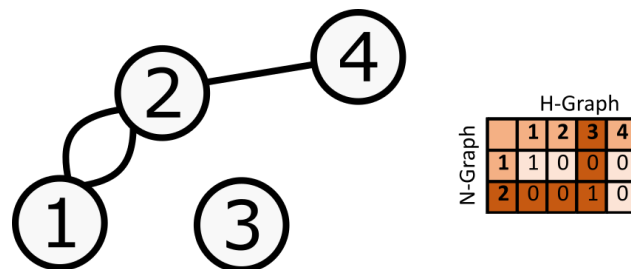


Abbildung 42: Alle Verbindungen nach 3 werden mit Verbindungen nach Zuweisung(3) = 2

Schritt 4

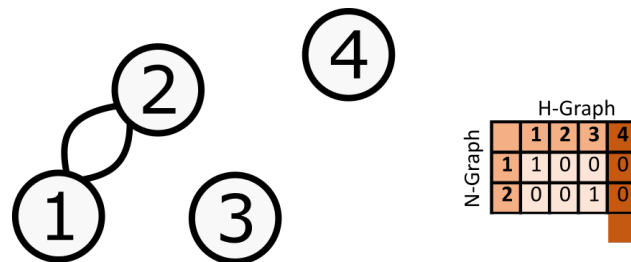


Abbildung 43: Alle Verbindungen nach 4 werden mit Verbindungen nach Zuweisung(4) = Keine

Schritt 5

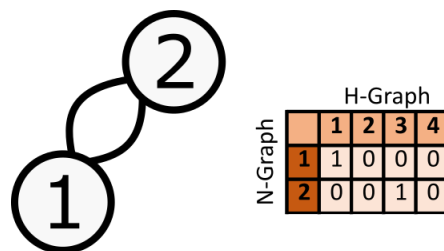


Abbildung 44: Alle Vertices die nicht in N vorkommen werden entfernt

Schritt 6

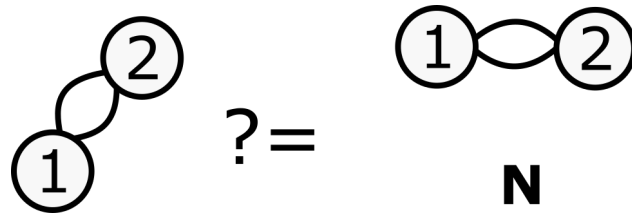


Abbildung 45: Sind die Geraphen gleich? Wenn ja: Zuordnung ist möglich, sonst nicht! In diesem Fall: Zuordnung möglich.

8 Umwandeln der Schaltung in eine Graph Datenstruktur

8.1 Begriffe

8.1.1 Richtungs-Gradient

Ist ein Konstrukt, welches eine Liste von besuchten Koordinaten mitführt. Bei jeder neu hinzugefügten Koordinate wird die Richtung der letzten N Koordinaten geprüft. Sollte diese Richtung durchschnittlich eine andere sein als jene zu vor wir eine Richtungsänderung erkannt!

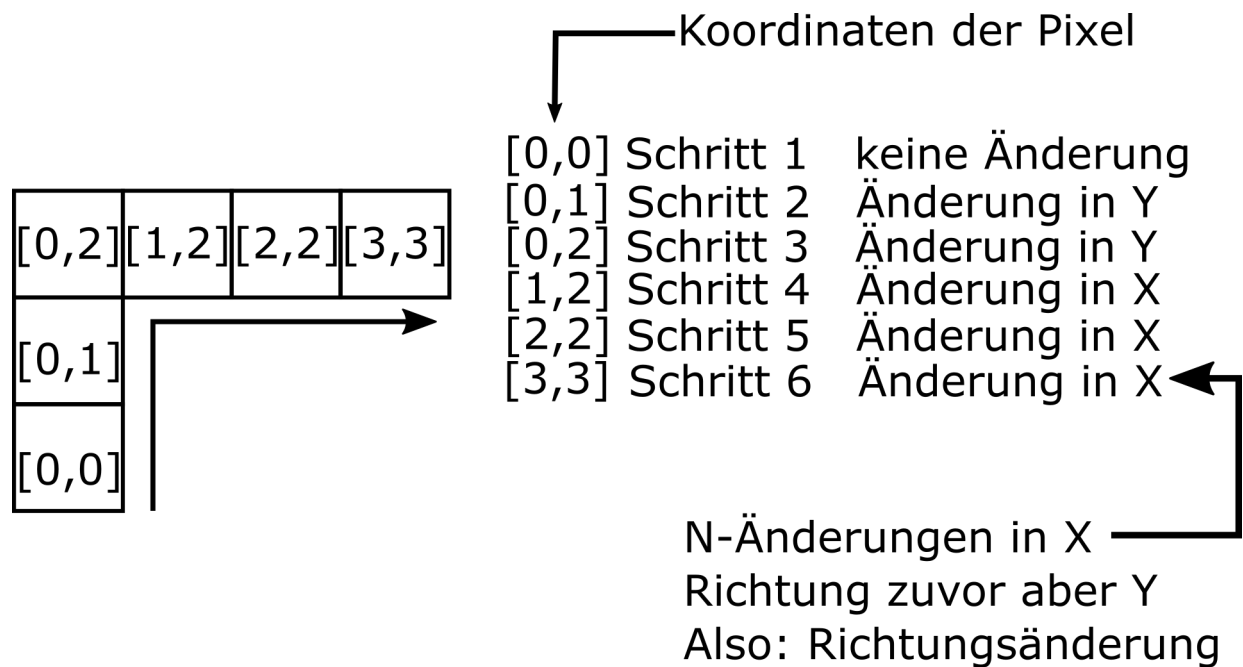


Abbildung 46: Beschreibung eines Richtungs-Gradienten

8.2 Problem

Für eine leichtere Verarbeitung/Bearbeitung sollte das Bild der Schaltung in eine Datenstruktur umgewandelt werden. Da ein Graph die in dem Bild der Schaltung enthaltene Information sehr gut speichert, wurde diese Datenstruktur gewählt. Für eine genauere Beschreibung siehe Kapitel zuvor.

8.3 Idee

Mithilfe eines rekursiven Algorithmus sollen die schwarzen Pixel in der Schaltung, Pixel für Pixel verfolgt werden.

So sollen alle Eckpunkte, Endpunkte und Verzweigungen gefunden werden, in je einen Vertex umgewandelt und sinnvoll zu einem Graphen zusammengefügt werden. Jeder der drei Fälle soll dabei ein Vertex mit je einer anderen Farbe sein.

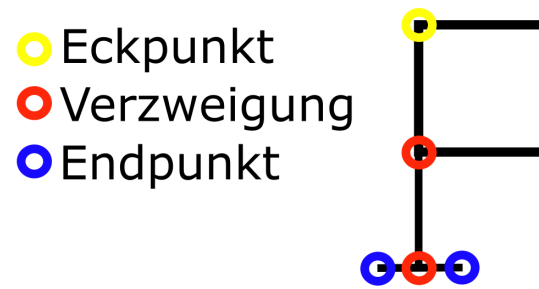


Abbildung 47: Unterschiedliche Typen von Vertices

8.4 Umsetzung

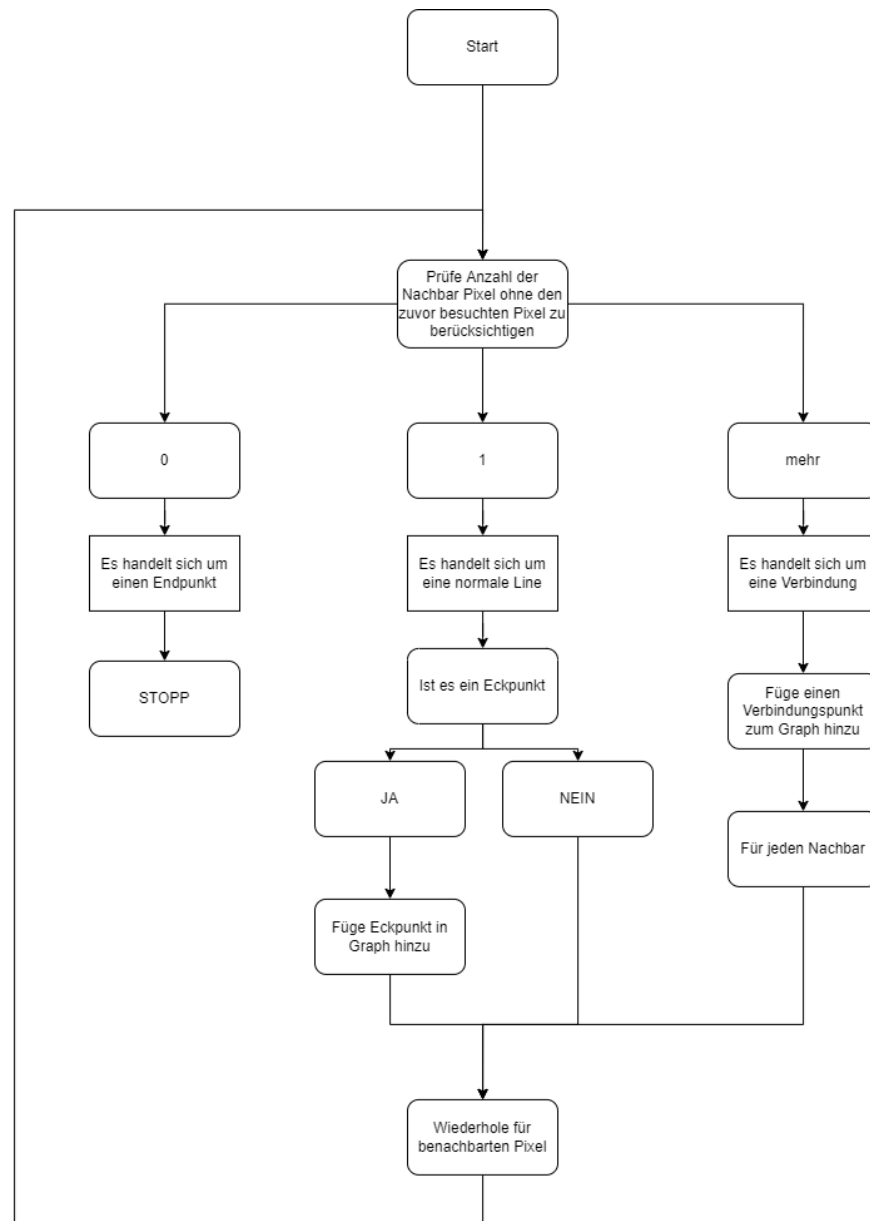


Abbildung 48: Blockschaltbild des rekursiven teils des Algorithmus

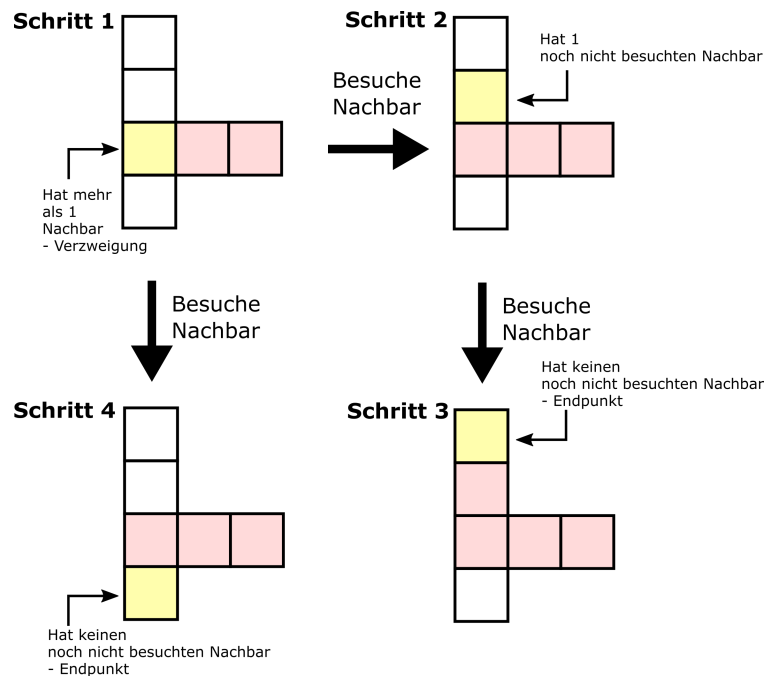


Abbildung 49: Beispiel Szenario

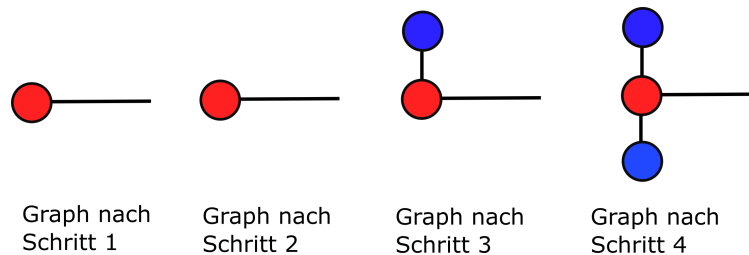


Abbildung 50: Der von oben generierte Graph bei den einzelnen Zwischenschritten

8.5 Algorithmus

Der Graph wird nach folgendem Algorithmus generiert.

Als Input für den Algorithmus muss ein, laut vorherigen Kapiteln bearbeitetes Bild, als 2D-Array übergeben werden. Richtungsänderungen werden mithilfe eines Richtungs-Gradienten erkannt. Für eine Erklärung siehe Richtungs-Gradient zuvor.

```

erstelle ein leeres Graph Objekt
while - es existieren noch schwarze Pixel im Bild
    finde die Koordinaten eines zufälligen, schwarzen Pixels
    führe "generatePartGraph" mit gefundenen Koordinaten aus
    führe den generierten Graphen mit dem zuvor erstellen zusammen
    färbe alle besuchten Pixel weiß
  
```

```

subroutine "generatePartGraph"( startPoint ):
  
```

```

    finde alle schwarzen, benachbarten Pixel des gegebenen Startpunktes
  
```



```

je nach Anzahl der gefunden Pixel
  1 Nachbar:Endpunkt
    füge einen Vertex mit der blauen Farbe hinzu
  2 Nachbarn:Ein normales Verbindungsstück
    füge einen Vertex mit der pinken Farbe hinzu
  mehr als 2 Nachbarn:Eine Verzweigung
    füge einen Vertex mit der pinken Farbe hinzu

```

```

rufe die Subroutine "rekursiv" auf mit:
  currentPixel = startPoint
  lastPixel = [0,0]
  lastGraphNode = zuvor erzeugter Vertex
  directionGradient = neuer Richtungs Gradient

```

```

return erstellen Graph und besuchte Pixel

```

```

subroutine "rekursiv"(currentPixel, lastPixel, lastGraphNode, directionGradient)
  füge eines neuen Schritt zum Richtungs-Gradient hinzu

```

```

  prüfe ob der gerade besuchte Pixel bereits zuvor besucht wurde
    wenn ja ->
      verbinde den Verbindungs-Vertex mit dem letzt erstellten Vertex
      return

```

```

  markiere den gerade besuchten Pixel als besucht

```

```

  hole alle zum gerade besuchten Pixel, benachbarten Pixel
  jedoch nicht den zuvor besuchten Pixel

```

```

je nach Anzahl der Nachbarn:

```

```

  0 Nachbarn: Endpunkt
    füge einen Endpunkt-Vertex zum Graphen hinzu
    verbinde diesen mit dem zuletzt erstellen Vertex

```

```

  1 Nachbar: einfache Verbindungslinie
    prüfe ob der Richtungs-Gradient eine Ecke findet
      wenn ja ->
        füge einen Eckpunkt-Vertex zum Graphen hinzu
        verbinde diesen mit dem zuletzt erstellen Vertex
        rufe die Subroutine "rekursiv" auf mit:
          currentPixel = Nachbar-Pixel
          lastPixel = gerade besuchter Pixel
          lastGraphNode = gerade erstellter Vertex
          directionGradient = neuer Richtungs-Gradient

```

```

      wenn nein ->
        rufe die Subroutine "rekursiv" auf mit:
          currentPixel = Nachbar-Pixel
          lastPixel = gerade besuchter Pixel
          lastGraphNode = zuletzt erstellter Vertex
          directionGradient = Richtungs-Gradient

```

```

  mehr als 1 Nachbar:
    füge dem Graphen einen Verzweigungs-Vertex und
    verbinde diesen mit dem zuletzt erstellen Vertex

```

```
für jeden Nachbar:  
  rufe die Subroutine "rekursiv" auf mit:  
    currentPixel = Nachbar-Pixel  
    lastPixel = gerade besuchter Pixel  
    lastGraphNode = gerade erstellter Vertex  
    directionGradient = Richtungs-Gradient
```

9 Finden der Bauteile

9.1 Problem

Damit später die Bauteile in ein Simulationsprogramm eingefügt werden können, muss als erstes die Position der Bauteile innerhalb der Schaltung gefunden werden.

9.2 Idee

In dem Graph sucht man nach bestimmten Mustern, welche ein Bauteil darstellen. Dadurch dass in jedem Vertex die original Koordinaten gespeichert sind, kann dann die Positon des jeweiligen Bauteils im Ausgangsbild gefunden werden.

9.3 Umsetzung

Im Graphen, ist jedes Bauteil relativ leicht anhand seines Musters zu identifizieren.

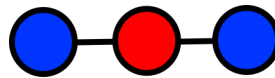


Abbildung 51: Muster für ein Gnd Symbol im Graphen

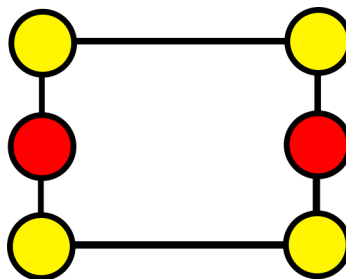


Abbildung 52: Muster für einen Widerstand im Graphen

Mithilfe der Muster-Findungs-Funktion die unsere Graph Bibliothek zur Verfügung stellt, können die Muster dann sehr leicht gefunden werden. Da in jedem Vertex dann noch seine Koordinaten im Bild gespeichert sind, können dann sehr leicht sogenannte Bounding Boxes erstellt werden. Wichtig ist, dass in diesem Schritt lediglich die Position der Bauteile gefunden, nicht die Art des Bauteils.

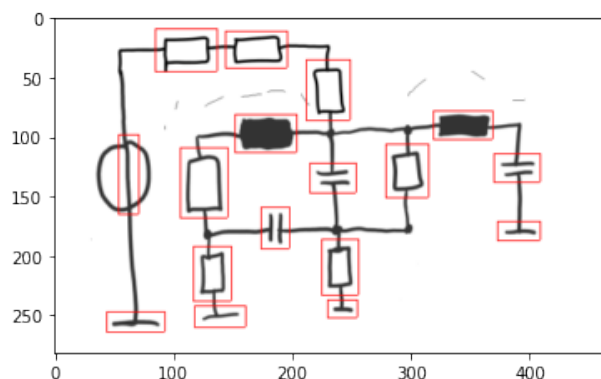


Abbildung 53: Beispielschaltung mit eingezeichneten Boundingboxen

Wichtig: Um manche Bauteile (besonders Kondensatoren) zu erkennen, muss der Graph, bevor Muster gefunden werden können, noch modifiziert werden. Ein Kondensator würde ohne diese Modifikation wie 2 Ground Symbole aussehen.

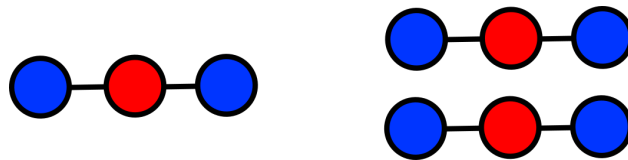


Abbildung 54: Muster eines Ground Symbols(links) und eines Kondensators(rechts)

Im Falle eines Kondensators wird geprüft, ob zwei Ground Symbole direkt gegenüber voneinander stehen. Sollte dies der Fall sein, wird der Graph so verändert, dass ein Kondensator eindeutig identifizierbar ist.

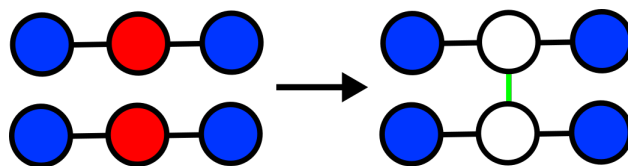


Abbildung 55: Die Veränderung des Kondensator Musters

10 Bauteilklassifizierung

10.1 Problem

Um die Schaltung zu digitalisieren, müssen alle Bauteile, nachdem deren Position gefunden worden ist, noch richtig erkannt werden.

10.2 Idee

Um die Bauteile zu erkennen soll ein "Convolutional Neural Network" verwendet werden.

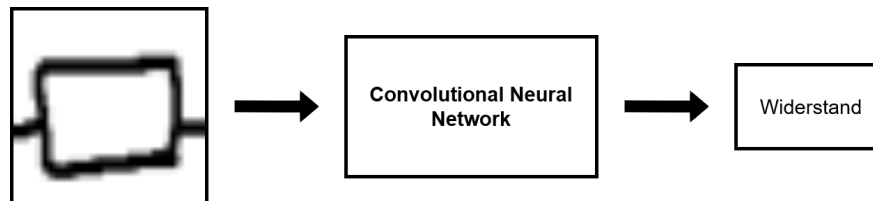


Abbildung 56: Funktion CNN

Diesem Netzwerk wird das Bild eines Bauteiles übergeben und als Antwort erhält man um welches Bauteil es sich am wahrscheinlichsten handelt.

10.3 Neuronales Netzwerk

Ein Neuronales Netzwerk ist ein Algorithmus der dem Menschlichem Gehirn nachempfunden ist. Solche Netzwerke können verwendet werden, um Muster in Daten zu finden, wie zum Beispiel Gegenstände auf Bildern zu erkennen.

10.3.1 Aufbau

Ein Neuronales Netzwerk besteht aus Neuronen, die in mehreren Schichten (Layer) angeordnet sind. Jedes Netzwerk hat am Anfang einen Input Layer sowie am Ende einen Output Layer. Zwischen diesen beiden Layer können einer oder mehrere Hidden Layer liegen.

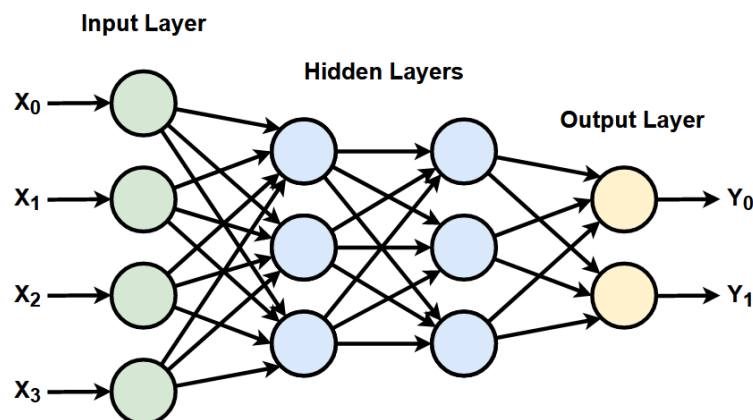


Abbildung 57: Struktur eines Neuronalen Netzwerkes

Jedes Neuron enthält eine Nummer auch Aktivierung genannt.

Um Bauteile einer Schaltung zu erkennen, würde die Aktivierung der Neuronen im input Layer jeweils der Pixelwert eines Bildes sein.

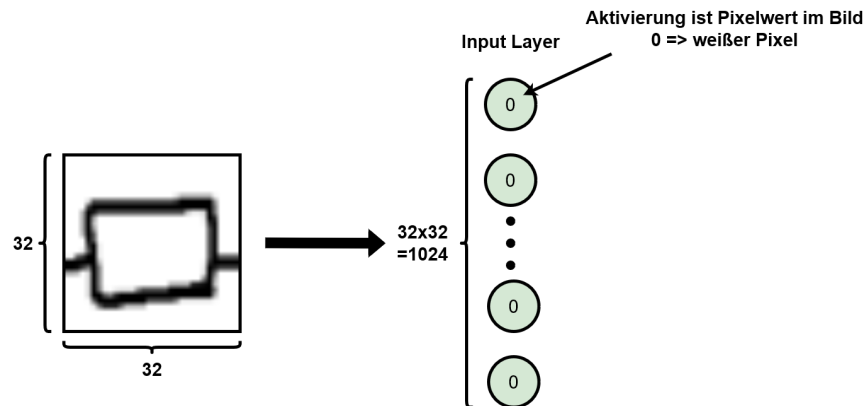


Abbildung 58: Input Layer eines Neuronales Netzwerkes

Die Länge des Output Layers wäre die Anzahl der verschiedenen Bauteile die dieses Netzwerk erkennen kann. Dabei ist die Aktivierung eines Neurons am Ausgang die jeweilige Wahrscheinlichkeit, dass auf dem bild das jeweilige Bauteil abgebildet ist.

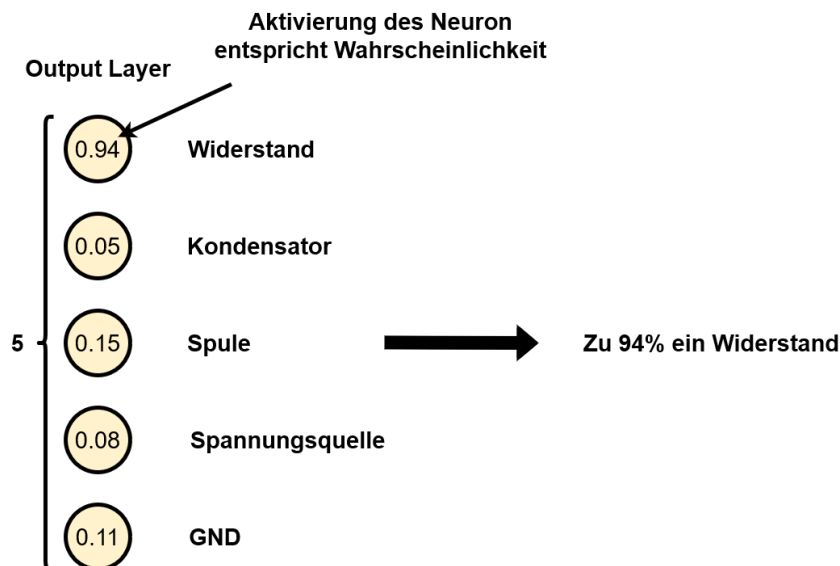


Abbildung 59: Output Layer eines Neuronales Netzwerkes

Alle Neuronen in einem Layer sind mit jeweils allen Neuronen aus dem vorherigen Layer über Gewichte (weights) verbunden und besitzen einen Schwellwert (bias).

Um die Aktivierung eines Neurons zu berechnen wird zuerst die gewichtete Summe berechnet. Für die gewichtete Summe z wird jeder Eingang x mit seinem jeweiligen Gewicht w multipliziert und diese werden mit dem bias des

Neurons addiert. Die gewichtete Summe wird zum Schluss in eine Funktion (Activation function) gegeben, der Funktionswert entspricht der Aktivierung.

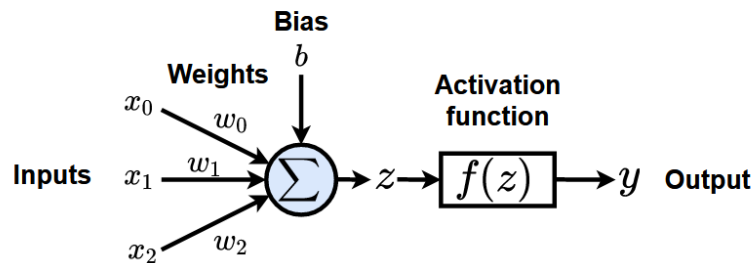


Abbildung 60: Aktivierung eines Neurons

In einem Neuronalen Netzwerk sind die Eingänge x jeweils die Aktivierung aus dem vorherigen Layer.

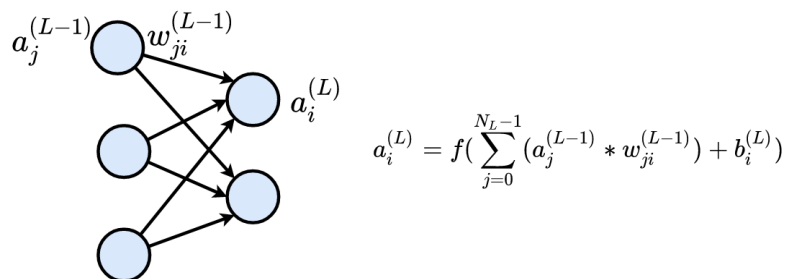


Abbildung 61: Berechnung der Aktivierung in einem Netzwerk

10.3.2 Trainieren eines Neuronales Netzwerkes

Die Gewichte und Schwellwerte eines Neuronales Netzwerkes können mithilfe eines Algorithmus "erlernt" werden. Dafür wird eine große Anzahl an Trainingsdaten benötigt. Das sind Daten bei denen der gewünschte Ausgabewert bereits bekannt ist.

Zuerst werden die Gewichte und Schwellwerte zufällig gewählt. Danach wird die Ausgabe des Netzwerkes für ein Trainingsbild mit dem gewünschten Ausgabewert verglichen und es kann somit festgestellt werden wie gut das Netzwerk funktioniert. Die Gewichte werden daraufhin so verändert, dass die Ausgabe des Netzwerkes sich dem gewünschten Ausgabewert annähert. Dies wird für alle Trainingsdaten so lange wiederholt, bis Ausgabe des Netzwerkes mit den gewünschten Ausgabewerten annähernd übereinstimmt.

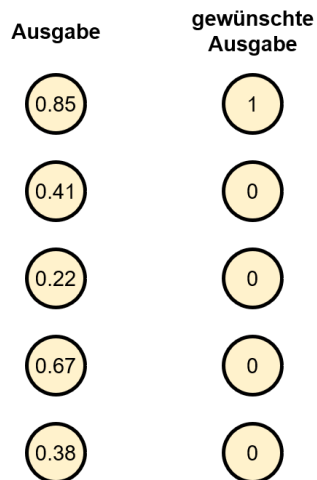


Abbildung 62: Ausgabe und gewünschte Ausgabe

10.4 Convolutional Neural Network

Ein Convolutional Neural Network besitzt im gegensatz zu einem Neuronalen Netzwerk mehr verschiedene Layer.

10.4.1 Filter

Bei einem Convolutional Layer werden ein oder mehrere Filter verwendet.

Diese Filter werden dabei über das Bild geschoben und die darunter liegenden Werte werden mit den Filterwerten multipliziert und addiert. Der resultierende Wert wird daraufhin in einer sogenannte "Feature map" gespeichert.

10.4.2 Beispiel Filter

Bei diesem Beispiel wird ein Filter verwendet um eine spezielle Ecke zu erkennen. Je höher der Wert in der Feature map desto höher ist die Übereinstimmung.



Abbildung 63: Beispiel Filter zur Erkennung von Ecken

10.4.3 Convolutional Layer

Ein Convolutional Layer besteht meist aus mehreren Filtern die eine Vielzahl an Feature maps erzeugen.

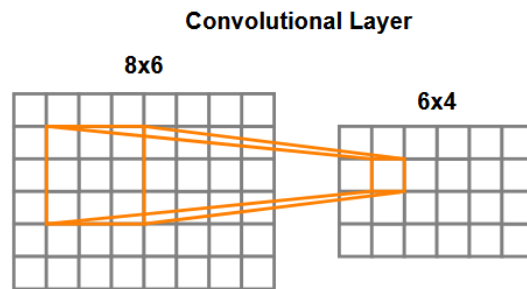


Abbildung 64: Beispiel Convolutional Layer mit einem Filter

10.4.4 Pooling Layer

Ein weiterer Layer ist der Pooling Layer. Um die Dimensionen der Feature maps zu reduzieren werden pooling Layer verwendet. Dies reduziert den Rechenaufwand und somit die Zeit ein Netzwerk zu trainieren.

Ein Beispiel für einen pooling Layer ist max Pooling.

10.4.5 Beispiel Max pooling

Bei diesem Beispiel ist die Feature map am Anfang eine 4x4 Matrix. Der Filter hat eine Größe von 2x2 und wird immer um 2 Felder weiterbewegt. Für jede Region wird nur der maximale Wert übernommen und in eine neue Matrix eingetragen.

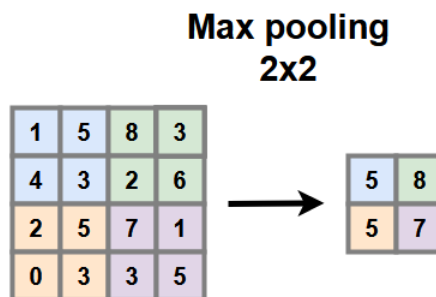


Abbildung 65: Max pooling

10.4.6 Fully Connected Layer

Zum Schluss werden noch, wie bei einem Neuronales Netzwerk, vollständig verbundene Layer verwendet um aus den Feature maps die endgültigen vorhersagen zu berechnen.

Fully Connected Layer

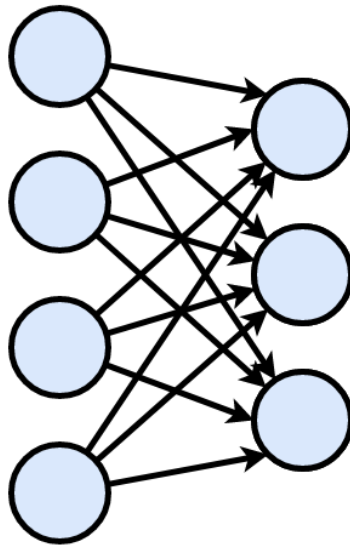


Abbildung 66: Fully Connected Layer

10.5 Umsetzung

Das Convolutional Neural Network wurde mit der open-source Library Tensorflow implementiert.

Die Position der Bauteile ist bereits bekannt. Die davor erstellten Bounding Boxen werden aus dem Binärbild ausgeschnitten und auf eine Größe von 32x32 Pixel skaliert. Die einzelnen Pixelwerte des Bildes werden davor noch durch 255 dividiert damit jeder Wert zwischen 0 und 1 liegt. Jedes davor gefundene Bauteil wird so klassifiziert.

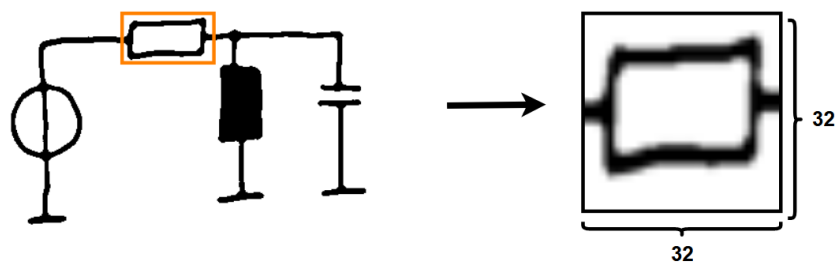


Abbildung 67: Fully Connected Layer

11 Rotationserkennung

11.1 Problem

Nachdem alle Bauteile klassifiziert wurden, muss für jedes Bauteil noch festgestellt werden in welche Richtung (horizontal oder vertikal) es gedreht ist, damit das Bauteil in LTspice richtig gezeichnet werden kann.

11.2 Idee

11.3 Rotationserkennung mit Neuronalem Netzwerk

Dass Neuronale Netzwerk, welches darauf trainiert wurde, die Bauteile zu klassifizieren soll zusätzlich noch die Rotation jedes Bauteils herausfinden.

Diese Methode der Rotationserkennung war jedoch sehr ungenau, weshalb eine andere Methode gewählt wurde.

11.4 Umsetzung

Der Funktion, welche die Rotation erkennen soll, wird eine Liste mit allen Vertices des Bauteils übergeben.

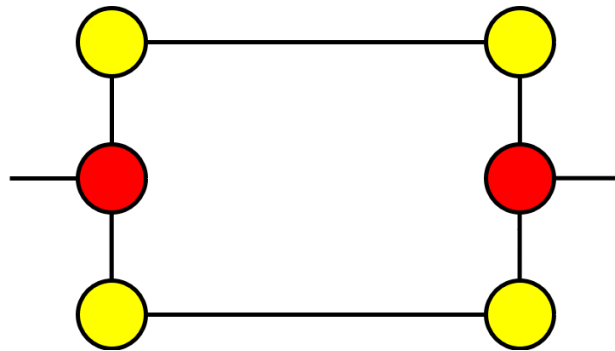


Abbildung 68: Graph eines Widerstandes

Aus dieser Liste werden alle Intersection Vertices herausgesucht. Danach wird jeweils der Abstand der beiden Vertices in X-Richtung und Y-Richtung berechnet. Ist der horizontale Abstand größer, ist das Bauteil horizontal gezeichnet ansonsten vertikal.

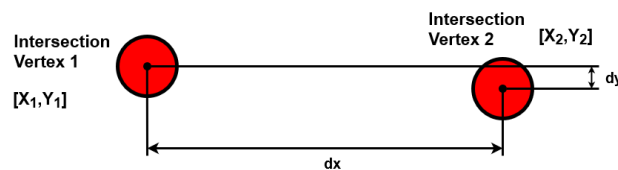


Abbildung 69: Abstände zwischen den Vertices

12 Umstrukturieren des Graphen

12.1 Problem

Um den Graphen nun besser in ein von Simulatoren verständliches File umzuwandeln, muss er zuvor noch umstrukturiert werden. Als Erstes müssen alle Koordinaten in relative Werte umgewandelt werden. Dies ist nötig, da die Bauteile im Simulatorenprogramm eine fixe Größe haben. Ist nun allerdings eine Schaltung sehr klein gezeichnet, wäre eine Umwandlung nicht möglich!

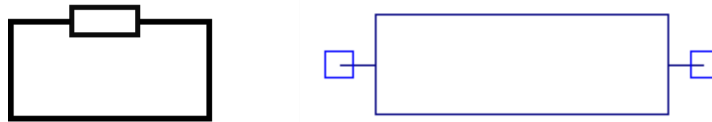


Abbildung 70: Eine solche Umwandlung wäre nicht möglich, da bereits der Widerstand größer ist als die gesamte Schaltung

Auch müssen alle Bauteile noch neu positioniert werden, um geradlinige Linienführung garantieren zu können.

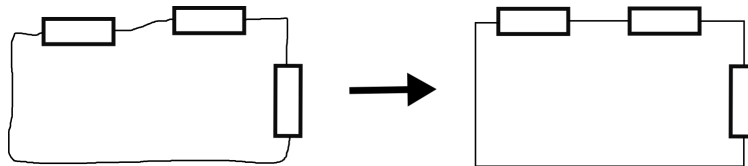


Abbildung 71: Problem mit nicht geradlinigen Linienführungen

12.2 Idee

Mithilfe eines Algorithmus soll der Graph so umstrukturiert werden, dass alle Bauteile zu je einem Vertex zusammengefasst werden.

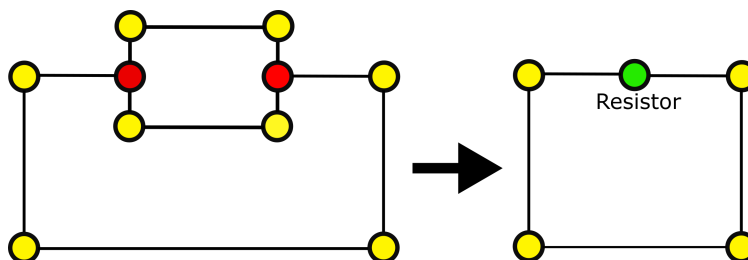


Abbildung 72: Umstrukturierung des Graphen

Die Umwandlung in relativen Koordinaten soll geschehen, indem zuerst die durchschnittliche Länge der gezeichneten Widerstände ermittelt wird. Dann sollen alle Koordinaten so skaliert werden, dass die gezeichneten Widerstände genau so groß sind wie die Widerstände im Simulatorenprogramm.

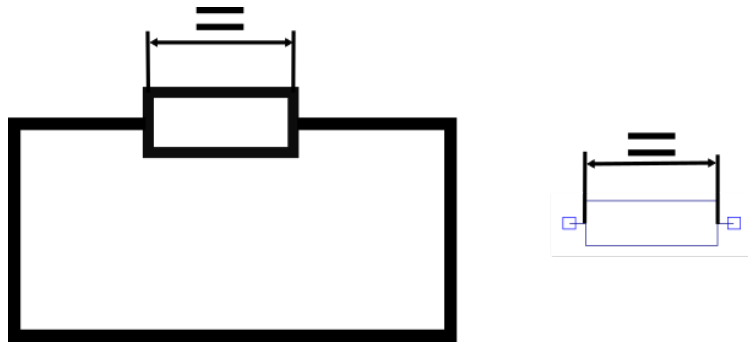


Abbildung 73: Skalieren der Koordinaten

Mithilfe eines weiteren rekursiven Algorithmus, welcher Bauteil für Bauteil alle Bauteile besucht, sollen jene dann, wie in dem "Problem" bereits geschrieben, gerade gerichtet werden.

12.3 Umsetzung

12.3.1 Konvertierung zu relativen Koordinaten

Zuerst werden die Bauteilinformationen wie Rotation, Art des Bauteils und Bauteil-Vertices genutzt, um die durchschnittliche Länge der gezeichneten Widerstände zu ermitteln. Dann werden alle Koordinaten, sowohl X als auch Y durch diese Länge dividiert und mit der Länge des Widerstandes im Simulatoren-Programm multipliziert.

12.3.2 Umstrukturieren des Graphen

Für jedes vom vorherigen Schritt gefundenes Bauteil wird ein neuer, grüner Vertex in den Graphen eingefügt.

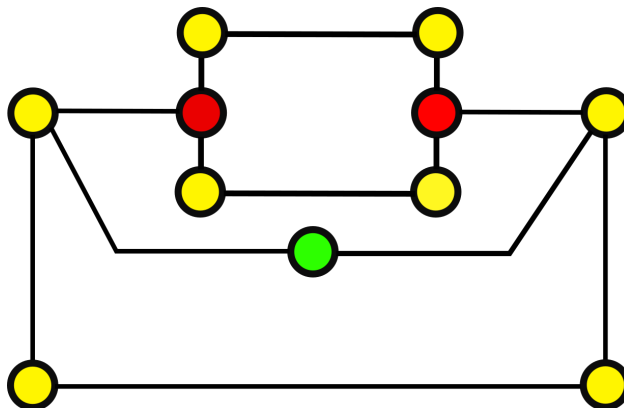


Abbildung 74: Einfügen eines grünen Vertex für jedes Bauteil

Anschließend werden die eigentlichen Vertices des Bauteils entfernt. Dies alles geschieht mithilfe der "group" Methode, welche unsere Graph-Bibliothek für ein Graph Objekt zur Verfügung stellt.

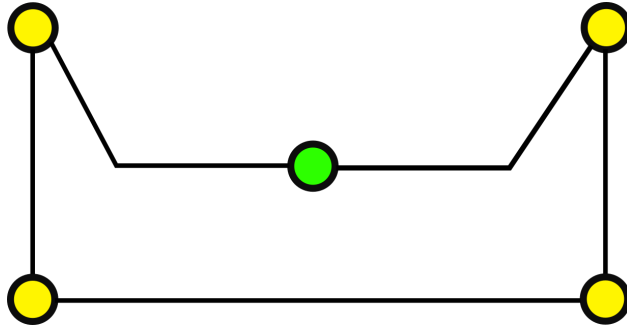


Abbildung 75: Entfernen der übrigen Bauteil-Vertices

Dann werden für die Verbindungen zwischen den Bauteilen noch neue Verbindungs-Vertices eingefügt.

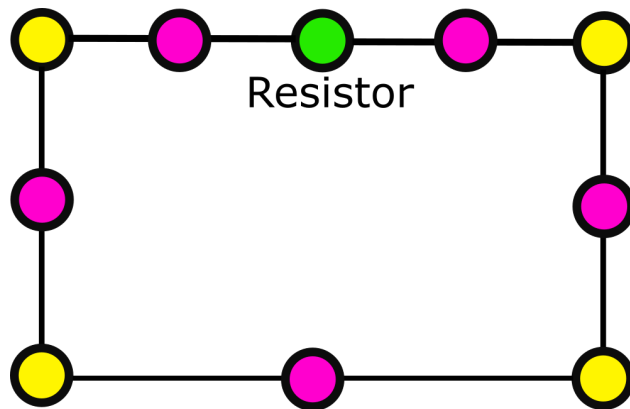


Abbildung 76: Einfügen von Verbindungs-Vertices

Die Bauteile werden dann als extra Vertices aus dem Graphen genommen.

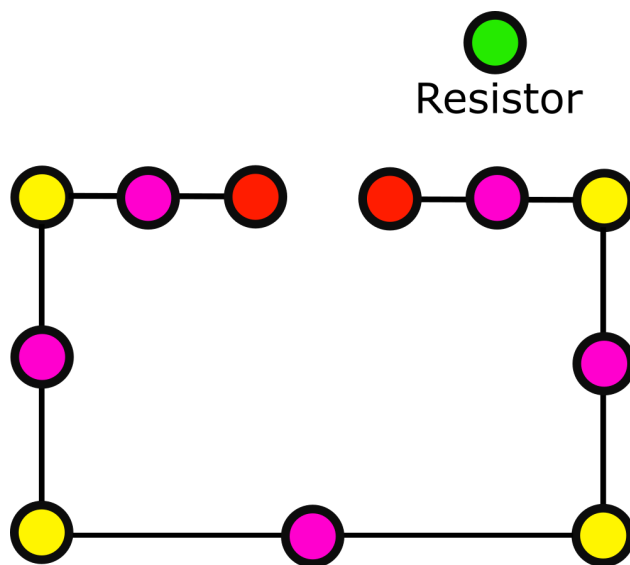


Abbildung 77: Bauteile ausgliedern

12.3.3 Bauteile gerade richten

Beginne bei einem beliebigen Vertex. Besuche dessen Nachbarn. Vergleiche die Koordinaten der Nachbarn mit den Koordinaten des gerade besuchten Vertex.

$$dX = X1 - X2$$

$$dY = Y1 - Y2$$

ist dX größer, so setze die Y Koordinate des Nachbars = der Y Koordinate des gerade besuchten Vertex
bei dY größer, setze die X Koordinate des Nachbars = der X Koordinate des gerade besuchten Vertex

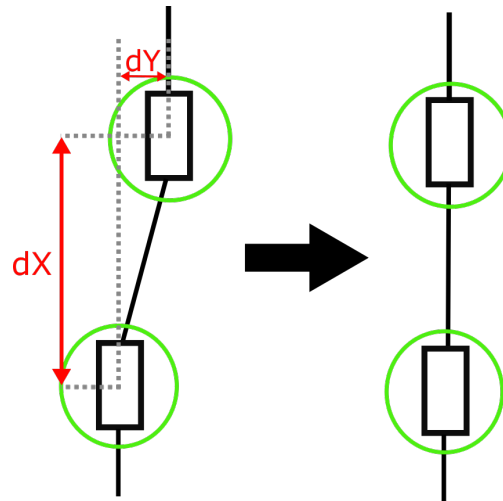


Abbildung 78: dY ist kleiner und wird somit zu 0 gemacht

Wiederhole dies für alle Nachbarn bis alle Vertices einmal besucht wurden

12.3.4 Beschreibung des Algorithmus

finde einen Eck-Vertex als Startpunkt

subroutine "traversal"(currentVertex, lastVertex)

 berechne $X1 - X2$ von dem gerade besuchten und zuletzt besuchten Vertex

 berechne $Y1 - Y2$ von dem gerade besuchten und zuletzt besuchten Vertex

 wenn $X1 - X2$ kleiner

 setze X Position vom gerade besuchten Vertex

 zur X Position des zuletzt besuchten Vertex

 sonst

 setze Y Position vom gerade besuchten Vertex

 zur Y Position des zuletzt besuchten Vertex

hole alle Nachbarn des gerade besuchten Vertex

für jeden Nachbar

 wenn Nachbar nicht gleich zuletzt besuchter Vertex

 rufe Subroutine "traversal" auf mit:

 currentVertex = Nachbar

 lastVertex = currentVertex

12.3.5 Bauteile gerade richten (nicht genutzt)

Die Idee war es einfach alle Koordinaten auf einen gewissen Wert zu runden, so würde [175,142] zu [180,140] werden.

Problem: Schaltungen werden unterschiedlich groß gezeichnet. Somit müssten die Koordinaten auch auf unterschiedliche Werte gerundet werden. Dies ist allerdings kaum machbar.

13 Output File erstellen

13.1 Problem

Der fertig bearbeitete Graph muss nun in ein vom Simulator "LT-Spice" verständliches Format umgewandelt werden.

13.2 Idee

Das Format, indem LT-Spice Schaltungen gespeichert werden, ist ein einfaches ASCII-Format. Kann also relativ einfach erstellt werden. Für mehr Information über die Syntax des Files, siehe LT-Spice Syntax

13.3 LT-Spice Syntax

Generelle Informationen über die Version und der Größe des LT-Spice Files

```
Version 4
SHEET 1 1000 1000
```

Beispiel für die Platzierung eines Bauteils samt Beschriftung -> Art des Bauteils ("cap","resistor",...) -> Die Drehung des Bauteils. Entweder R0 oder R90 -> Sichtbarer Name des Bauteils

```
SYMBOL <bauteil> <x-pos> <y-pos> <R90/R0>
SYMATTR InstName <name>
```

Platzierung eines Ground Symbols

```
FLAG <x-pos> <y-pos> 0
```

Platzierung eines Kabels um Bauteile zu verbinden

```
WIRE <x-pos-von> <y-pos-von> <x-pos-to> <y-pos-to>
```

13.4 Umsetzung

Für jedes Bauteil muss festgestellt werden, an welchen Koordinaten es mit dem Bauteil verbunden ist.

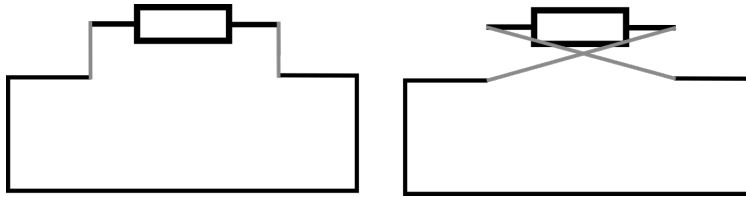


Abbildung 79: Die zwei Möglichkeiten, ein Bauteil anzuschließen

Als Erstes bekommt dabei jeder Anschluss eine Nummer zugeordnet.

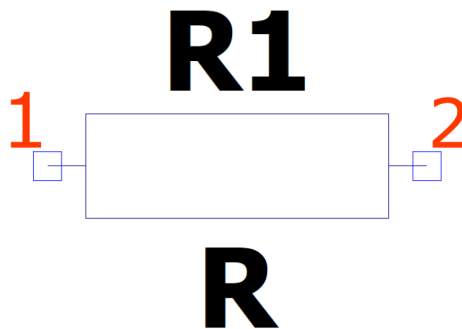


Abbildung 80: Nummerierte Anschlüsse an dem Widerstand

Die Reihenfolge wird dabei bestimmt durch den Abstand der linken oberen Ecke zu dem jeweiligen Anschluss.

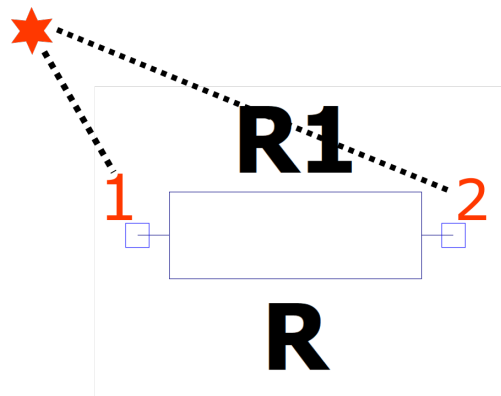


Abbildung 81: Abstand, kleinster zu größter aufsteigend

Das Selbe wird mit der gezeichneten Schaltung gemacht.

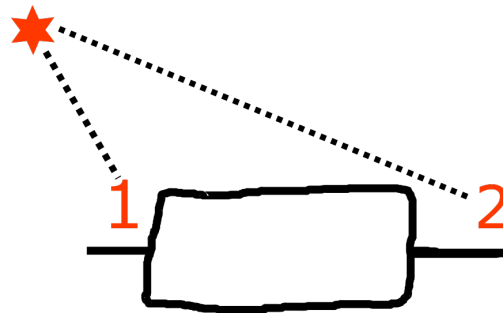


Abbildung 82: Abstand, kleinster zu größter aufsteigend

Durch diese Information können dann selbe Nummern zusammen geordnet werden und richtig verbunden werden. Speichere diese Information in den einzelnen Bauteil-Vertices.

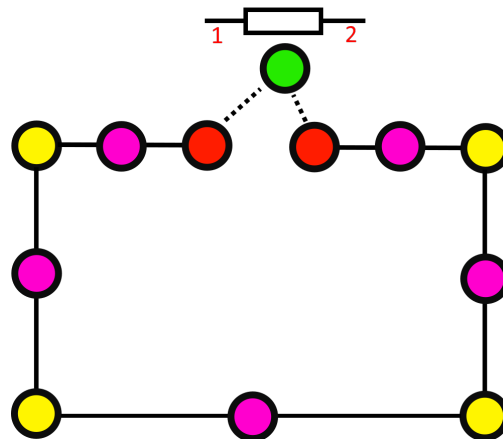


Abbildung 83: Graph mit Verbindungen gespeichert. In diesem Fall durch die gepunkteten Linien dargestellt

Für jeden Bauteil-Vertex füge den passenden LT-Spice Code in ein Output-File hinzu. Die Koordinaten entsprechen dabei den in dem Vertex Gespeicherten.

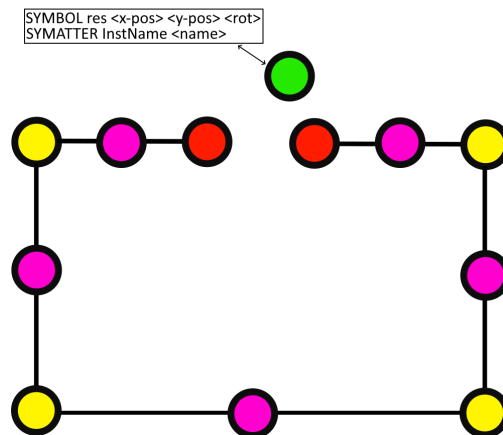


Abbildung 84: Widerstand wird als Text in das Output-File gespeichert

Für jeden Verbindungs-Vertex, füge den WIRE LT-Spice Code in das Output-File hinzu.

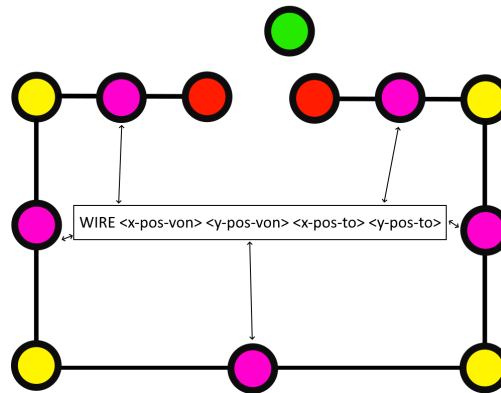


Abbildung 85: Verbindungen zwischen den Bauteilen werden in das Output-File geschrieben

Problem: Treffen 2 Bauteile direkt aufeinander

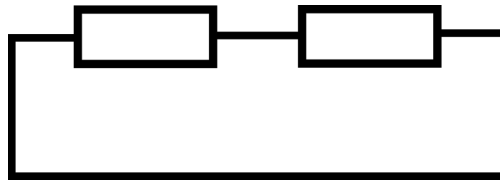


Abbildung 86: Beispiel für eine problematische Schaltung

Eine solche Schaltung würde einen solchen Graphen erzeugen.

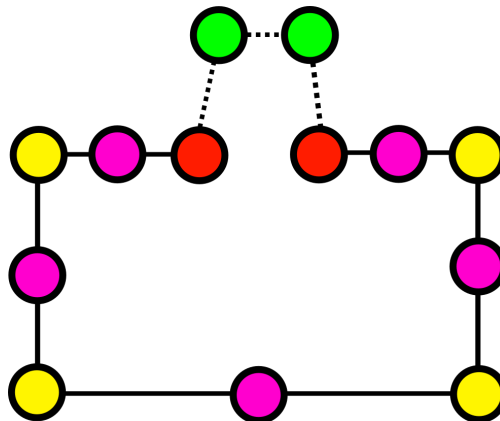


Abbildung 87: Gepunktete Linien stellen zuvor genannte Verbindungen dar

Mit der Beschreibung von oben hätte eine solche Schaltung jedoch keine Verbindung zwischen den beiden Widerständen, da ja dort kein Verbindungs-Vertex ist.

Lösung:

Sollte einer der Vertices mit denen ein Bauteil verbunden wird, ein weiterer Bauteil-Vertex sein, füge ein WIRE vom Anschluss des Bauteils bis zur Mitte, zwischen den beiden Bauteilen ein!



Abbildung 88: Schritt 1

Das selbe wird auch für den zweiten Widerstand gemacht.

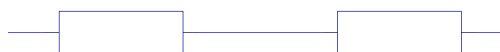


Abbildung 89: Schritt 2

Die Lösung mit einer Linie nur bis zu Mitte der beiden Bauteile, wurde gewählt, da ein Bauteil somit nicht wissen muss, welches Bauteil sein Nachbar ist.

14 Abbildungsverzeichnis

1	Blockschaltbild für unser Projekt	5
2	Bild öffnen	7
3	Bild im File Explorer auswählen	8
4	Bild zuschneiden	8
5	Binärbild erstellen	9
6	Binärbild erstellen	10
7	Bild umwandeln	11
8	Schaltung in LTSpice	11
9	Binärbild und gewünschtes Ergebnis	12
10	Problem bei Spulen	12
11	Lösung für Spulen	13
12	Linien Dicke in einer Spalte	13
13	Funktion "erode" Beispiel 1	14
14	Funktion "erode" Beispiel 2	14
15	Entfernen der Schaltung bis auf Spulen	14
16	Differenz zwischen den beiden Bildern	15
17	Bild mit einer Linienbreite von einem Pixel	15
18	Beispiel eines zu Entfernendes Muster	16
19	Entfernen eines Musters	16
20	Beispiel für einen Graphen	17
21	Bestandteile eines Graphen	17
22	Strukturell gleiche Graphen, welche sich lediglich durch die Farbe unterscheiden	17
23	Nachbar-Vertices	18
24	Vertices-Gruppieren	18
25	Zwischen-Vertices	19
26	Muster-Finden	19
27	Beispiel für eine Inzidenzmatrix	20
28	Beispiel für eine von mehreren möglichen Zuordnungen. Die jeweiligen Zuordnungen sind gepunktet dargestellt	20
29	Beispiel für 2 Graphen	21
30	Leere Matrix für die oben gezeigten Beispiele	21
31	In diesem Fall ist Vertex 1 in N, Vertex 2 in H zugeordnet	21
32	Beispieldurchgang für einen stark vereinfachten Graphen. In diesem Beispiel beginnt der Algorithmus mit einer ganz leeren Matrix anstelle einer bereits zum Teil gefüllten! Es wird dementsprechend also auch Schritt 2) ausgelassen	22
33	Beispiel für Überprüfung. Die Zuordnung ist dabei gepunktet	23
34	Suche den zugeordneten Vertex für 1	23
35	Suche die Nachbarn (hier grau)	23
36	Zuordnung von 1 auf 2 ist in diesem Fall somit möglich!	24
37	1 und 1 werden nie zuordenbar sein, da sie unterschiedliche Farbe haben und H1 nur einen Nachbar hat, N1 aber mindestens zwei benötigt	24
38	Beispiel für eine unmögliche Zuordnung	25
39	Beispiel Graphen	25
40	Alle Verbindungen nach 1 werden ausgetauscht mit Verbindungen nach Zuweisung(1) = 1	25
41	Alle Verbindungen nach 2 werden mit Verbindungen nach Zuweisung(2) = Keine	26
42	Alle Verbindungen nach 3 werden mit Verbindungen nach Zuweisung(3) = 2	26
43	Alle Verbindungen nach 4 werden mit Verbindungen nach Zuweisung(4) = Keine	26
44	Alle Vertices die nicht in N vorkommen werden entfernt	26

45	Sind die Geraphen gleich?Wenn ja: Zuordnung ist möglich, sonst nicht! In diesem Fall: Zuordnung möglich.	27
46	Beschreibung eines Richtungs-Gradienten	28
47	Unterschiedliche Typen von Vertices	29
48	Blockschaltbild des rekursiven teils des Algorithmus	30
49	Beispiel Szenario	31
50	Der von oben generierte Graph bei den einzelnen Zwischenschritten	31
51	Muster für ein Gnd Symbol im Graphen	34
52	Muster für einen Widerstand im Graphen	34
53	Beispielschaltung mit eingezeichneten Boundingboxen	34
54	Muster eines Ground Symbols(links) und eines Kondensators(rechts)	35
55	Die Veränderung des Kondensator Musters	35
56	Funktion CNN	36
57	Struktur eines Neuronalen Netzwerkes	36
58	Input Layer eines Neuronalen Netzwerkes	37
59	Output Layer eines Neuronalen Netzwerkes	37
60	Aktivierung eines Neurons	38
61	Berechnung der Aktivierung in einem Netzwerk	38
62	Ausgabe und gewünschte Ausgabe	39
63	Beispiel Filter zur erkennung von Ecken	39
64	Beispiel Convolutional Layer mit einem Filter	40
65	Max pooling	40
66	Fully Connected Layer	41
67	Fully Connected Layer	41
68	Graph eines Widerstandes	42
69	Abstände zwischen den Vertices	42
70	Eine solche Umwandlung wäre nicht möglich, da bereits der Widerstand größer ist als die gesamte Schaltung	43
71	Problem mit nicht geradlinigen Linienführungen	43
72	Umstrukturierung des Graphen	43
73	Skalieren der Koordinaten	44
74	Einfügen eines grünen Vertex für jedes Bauteil	44
75	Entfernen der übrigen Bauteil-Vertices	45
76	Einfügen von Verbindungs-Vertices	45
77	Bauteile ausgliedern	45
78	dY ist kleiner und wird somit zu 0 gemacht	46
79	Die zwei Möglichkeiten, ein Bauteil anzuschließen	49
80	Nummerierte Anschlüsse an dem Widerstand	49
81	Abstand, kleinster zu größter aufsteigend	49
82	Abstand, kleinster zu größter aufsteigend	50
83	Graph mit Verbindungen gespeichert. In diesem Fall durch die gepunkteten Linien dargestellt	50
84	Widerstand wird als Text in das Output-File gespeichert	50
85	Verbindungen zwischen den Bauteilen werden in das Output-File geschrieben	51
86	Beispiel für eine problematische Schaltung	52
87	Gepunktete Linien stellen zuvor genannte Verbindungen dar	52
88	Schritt 1	52
89	Schritt 2	52