Maturitätsarbeit an der Kantonsschule
Zürich Nord

Programmieren einer Software für das Erkennen der Farbe eines Rubik's Cubes

Leon Erzberger
M6d

Betreuungsperson
David Tyndall

Zürich, Dezember 2019

Quelle für Titelbild: [3]

Zusammenfassung

Ziel dieser Maturaarbeit war, ein möglichst gutes Programm für die Farberkennung eines Rubik's Cubes zu programmieren. Dafür wurden die beiden Farbräume RGB und HSV miteinander verglichen. Ausserdem wurde eine spezielle Methode für das Erkennen von schwarzen und weissen Feldern ausprobiert. Zum Schluss wurden dann noch zwei Methoden verglichen, die Korrekturen vornahmen, so dass jede Farbe exakt neunmal vorkommt. Das ausschliessliche Verwenden von Hue zur Unterscheidung der Farben, das Benützen der Methode, um Weiss und Schwarz auszusortieren, und das Verwenden einer der zwei Korrekturmethoden lieferte die besten Ergebnisse, mit durchschnittlich 8.5 Fehlern bei 54 Feldern.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Rubik's Cube . . . 1
- 1.2 Erste Idee . . . 2

2 Umsetzung und Resultate **3**
- 2.1 Farbräume . . . 3
  - 2.1.1 RGB . . . 3
  - 2.1.2 HSV . . . 4
  - 2.1.3 Anwendung im Code . . . 5
  - 2.1.4 Vergleich . . . 6
- 2.2 Weisse und schwarze Felder aussortieren . . . 8
  - 2.2.1 Problem . . . 8
  - 2.2.2 Lösung . . . 8
- 2.3 Korrekte Anzahl Felder pro Farbe . . . 10
  - 2.3.1 Nähe zu anderer Mitte . . . 10
  - 2.3.2 Distanz zu eigener Mitte . . . 11
  - 2.3.3 Vergleich . . . 12

3 Fazit **13**

4 Literatur **14**

1 Einleitung

In dieser Maturaarbeit wurde versucht, eine Software zu programmieren, welche die Farben eines Rubik's Cubes möglichst gut erkennen soll. Dabei wurden Probleme untersucht, die im Gebiet der Farberkennung auftreten können. Es werden verschiedene Lösungsansätze miteinander verglichen und es wird erklärt, wie sie im Code angewendet wurden.

1.1 Rubik's Cube

Um die Farben eines Rubik's Cubes zu erkennen, muss erst einmal klar sein, wie er überhaupt aufgebaut ist. Ein Rubik's Cube (im Folgenden Würfel genannt) besteht aus 26 Steinen. Diese sind in einer 3x3x3-Würfelform angeordnet, wobei der innere Stein fehlt. Die Steine sind mit sechs verschiedenen Farben so gefärbt, dass jede Seite des Würfels eine Farbe hat (vgl. Abbildung 1).
Dabei existieren insgesamt sechs Mittelsteine, auf jeder Würfelseite einer, die je eine einzige gefärbte Fläche haben. Diese Flächen werden von nun an „Mitte“ genannt. Diese Mitten haben jeweils eine der sechs Farben. Sie ändern beim Verdrehen des Würfels ihre relative Position zueinander nicht.
Neben den Mittelsteinen gibt es auch noch Seitensteine. Diese haben jeweils zwei oder drei gefärbte Flächen. So eine Fläche wird im Folgenden „Seite“ genannt.

Abbildung 1: Gelöster Rubik's Cube mit einer Farbe pro Seite. [2]

1.2 Erste Idee

Eine erste Idee ist, den Rubik's Cube immer an einem fixen Punkt zu befestigen, und ihn dann von zwei gegenüberliegenden Kameras zu fotografieren (vgl. Abbildung 2). So sind die Felder des Würfels immer an der gleichen Stelle im Bild, und es braucht keine Mustererkennung, um die Felder zu finden.

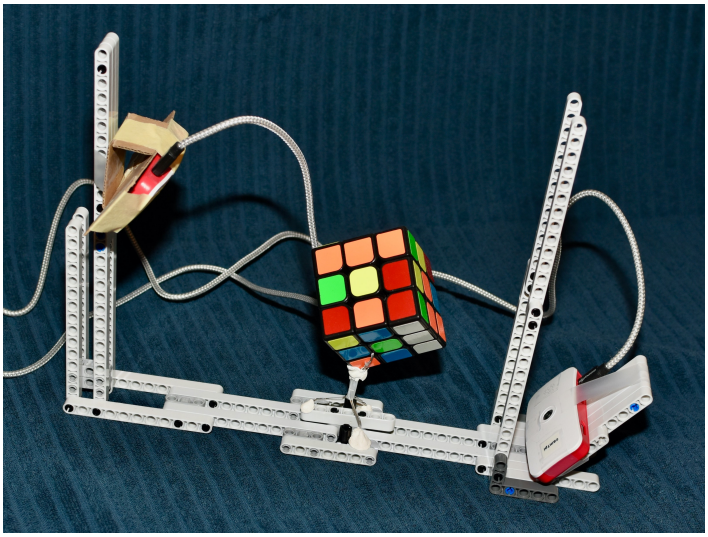

Abbildung 2: Aufbau, um Rubik's Cube immer vom gleichen Winkel aus zu fotografieren.

Um die Felder dann einer Farbe zuzuordnen, werden keine schon vorher fix definierten Farben verwendet, sondern es werden die Farbwerte der Mitten herausgelesen, und die restlichen Felder werden dann diesen zugeordnet. So wird vermieden, dass man bei jedem neuen Würfel, der andere Farben besitzt, wieder mühselig die richtigen Farbwerte suchen muss, um die Definitionsbereiche zu erstellen. Wenn ausserdem die Lichtverhältnisse mal etwas dunkler oder heller sind, wird dies auch automatisch miteinberechnet.

2 Umsetzung und Resultate

2.1 Farbräume

Um die Seiten zu den Mitten zuzuordnen, muss zuallererst festgelegt werden, wie die Farben der Felder aus dem Bild ausgelesen werden. Dafür muss man verstehen, wie Farben auf einem Bild überhaupt dargestellt werden. Dies geschieht auf einem Computer mithilfe eines Farbraumes. Ein Farbraum ist eine fest definierte Anzahl an Farben, die oft mithilfe von drei Variablen beschrieben werden. Eine Bilddatei enthält somit nur solche Farben, die durch eine Kombination der drei Variablen erstellt werden können. [5] Diese Variablenwerte können dann ohne Probleme ausgelesen und miteinander verglichen werden. Zwei der gängigsten Farbräume, der RGB- und der HSV-Farbraum, werden hier miteinander verglichen.

2.1.1 RGB

RGB ist der Standard-Farbraum für die digitale Bildwiedergabe. Er setzt sich aus drei Werten für Rot, Grün, und Blau zusammen. Die Werte gehen jeweils von 0 bis 255, wobei die Farbe bei 0 nicht vorhanden ist, und bei 255 mit voller Intensität leuchtet. (0, 0, 0) ist somit schwarz und (255, 255, 255) ist weiss. Der gesamte Farbraum lässt sich als Würfel in einem Koordinatensystem darstellen, wobei die Achsen jeweils die Intensität einer Farbe beschreiben (vgl. Abbildung 3).

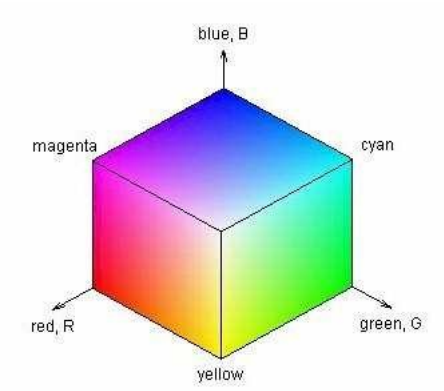

Abbildung 3: RGB-Farbraum als Würfel in einem Koordinatensystem mit Achsen für rot, grün und blau. [1]

Der RGB-Farbraum ist ein additiver Farbraum. Dies bedeutet, dass bei nichts angefangen wird, und je höher die Werte werden, desto mehr der entsprechenden Farbe vorhanden ist. [4]
Deshalb wird RGB auch bei vielen Arten von Beleuchtung verwendet, wie Fernseher und Computerbildschirme, die ihre Pixel mit roten, grünen und blauen LEDs beleuchten. Auch der Mensch erkennt Farben auf diese Weise, da im Auge Rezeptoren vorhanden sind, die empfindlich auf die Wellenlängen von Rot, Grün und Blau sind.

2.1.2 HSV

Der HSV-Farbraum setzt sich aus den Werten „Hue“, „Saturation“ und „Value“ zusammen. Hue ist ein Wert für den Farbton, und geht standardmässig von 0 bis 179. Ausserdem ist der Wertebereich von Hue kreisförmig. Das heisst, dass 0 und 179 nebeneinander sind und es somit kein Anfang oder Ende gibt.
„Saturation“ beschreibt die Sättigung und Intensität der Farbe und geht von 0 bis 255, wobei 0 keine Sättigung und somit Weiss bedeutet und bei 255 die Farbe vollends vorhanden ist.
„Value“ ist ein Mass für die Helligkeit und reicht ebenfalls von 0 bis 255. Wenn der Value bei null ist, dann spielt der Farbton keine Rolle, und die Farbe wird schwarz. Je höher der Value dann geht, desto heller wird die Farbe. [4]
Der HSV-Farbraum wird häufig in Form eines Zylinders visualisiert (vgl. Abbildung 4).

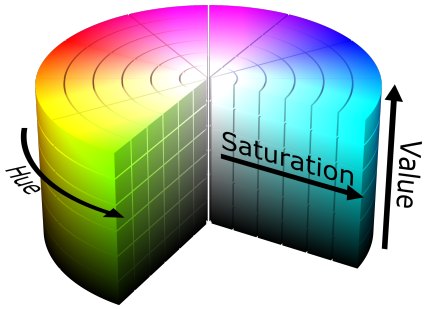

Abbildung 4: HSV-Farbraum als Zylinder mit Saturation als Radius, Value als Höhe und Hue als Winkel. [4]

2.1.3 Anwendung im Code

Das Farbenauslesen und das darauffolgende Zuordnen der Seiten wurde unter Verwendung von RGB folgendermassen im Code implementiert.
Zuerst werden für jedes Feld auf dem Würfel die RGB-Werte von 400 Pixeln in einem 20x20-Quadrat ausgelesen. Danach werden die Durchschnitte davon berechnet und in einer Liste gespeichert.

```
1  for rgb in range(3): # R-, G-, und B-Werte einzeln auslesen
2    col = 0
3    for m in range(20):
4      for n in range(20):
5        col += picture[x - 10 + m][y - 10 + n][rgb] # Werte
       einzelner Pixel addieren
6    avgcol[cubeside][tile][rgb] = int(col / 400) # Durchschnitt in
       Liste speichern
```

Listing 1: Auslesen der RGB-Werte der Felder des Rubik's Cubes.

Als nächstes werden die Seiten einer Mitte zugeordnet. Dafür wird in Zeile 2 eine Schleife initiiert, die für jede Seite die 6 Mitten durchgeht. Für jede Mitte wird damit der Abstand zur Seite berechnet.
Da RGB ein Farbraum mit kartesischen Koordinaten ist, kann der Abstand mit Hilfe des Satzes des Pythagoras berechnet werden. Somit kann man die Differenzen zwischen den R-, G-, und B-Werten einzeln berechnen und danach die Quadrate davon addieren. Dies geschieht in den Zeilen 5 und 6. Danach wird in Zeile 7 die Wurzel davon gezogen und das Ergebnis in einer Liste abgespeichert.
In den Zeilen 8-10 wird schliesslich evaluiert, welche Mitte am nächsten ist, und die Seite wird dieser zugeordnet.

```
1  col = 200000
2  for middle in range(6): # sechs Mitten durchgehen
3    dist = 0
4    for rgb in range(3):
5      dist += (avgcol[cubeside][tile][rgb] -
6           avgcol[middle][8][rgb]) ** 2 # Quadrate addieren
7    distances[cubeside][tile][middle] = mt.sqrt(dist)
8    if dist < col: # testen, ob Mitte am naechsten ist
9      col = dist
10     tilecol[cubeside][tile] = colors[middle]
```

Listing 2: Zuordnung der Seiten zu einer Mitte.

2.1.4 Vergleich

Um zu untersuchen, was die besten Ergebnisse liefert, wurden fünf verschiedene Ideen ausprobiert und verglichen:

1. Die RGB-Werte verwenden
2. Die HSV-Werte verwenden
3. Nur Hue und Saturation verwenden
4. Nur Hue und Value verwenden
5. Nur Hue verwenden

Diese fünf Methoden wurden alle auf zehn Bilder von Rubik's Cubes angewendet. Anschliessend wurde ermittelt, wie viele Fehler die Methoden im Schnitt ergaben.

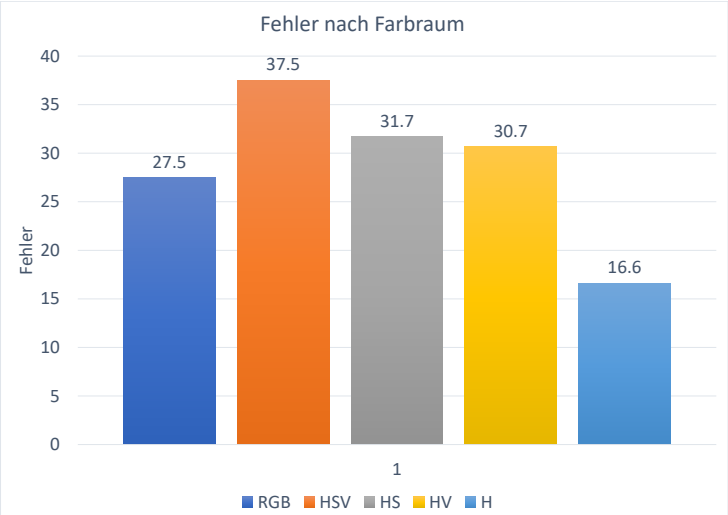

Abbildung 5: Durchschnittliche Anzahl Fehler für verschiedene Methoden.

Wie im obigen Diagramm zu sehen, lieferte die Methode 5 die besten Ergebnisse, mit durchschnittlich 16.6 Fehlern bei 54 Feldern auf einem Würfel. Am zweitbesten war die RGB-Methode, mit jedoch fast doppelt so vielen Fehlern. Danach kamen nahe beieinander die Methoden 3 und 4, und schliesslich mit am meisten Fehlern die Methode 2. Dass die Methoden, die Saturation

und/oder Value verwendeten, schlechter abschnitten, als wenn nur Hue verwendet wurde, war zu erwarten. Die Farben auf dem Würfel sind abgesehen von Weiss alle sehr satt und brillant. Wenn nun auf dem Bild eine Seite leicht überbelichtet oder schattig ist, dann verändert sich die Saturation oder der Value des Pixels. Somit werden dann zwei eigentlich identisch-farbige Felder als unterschiedlich angesehen, nur weil die Beleuchtung nicht identisch war. Diese Unterschiede werden bei Methode 5 nicht berücksichtigt, da nur die Hue-Werte verglichen werden. Auch die RGB-Methode erhält auf diese Weise Fehler. Da diese keine dedizierten Variablen für Sättigung und Helligkeit haben, verändern sich alle drei Variablen, wenn ein Feld eine andere Belichtung hat. Da sich diese Veränderung aber auf drei Variablen aufteilt, sind die einzelnen Werte nicht so weit entfernt, wie die von Saturation und Value. Und da die Quadrate der Entfernungen addiert werden, wirken sich zwei sehr falsche Werte mehr aus, als drei leicht falsche Werte.

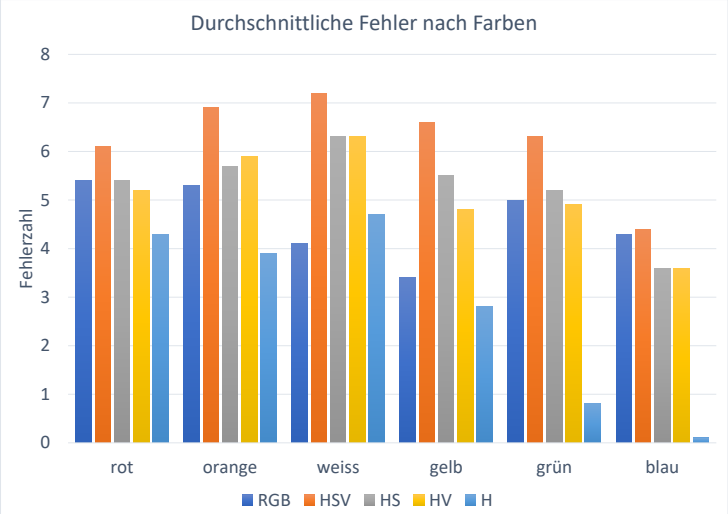

Abbildung 6: Durchschnittliche Anzahl Fehler für die verschiedenen Methoden, aufgeteilt nach den einzelnen Farben.

Wenn man die Fehler nun nach Farben aufteilt, sieht man, dass die Methode 5 bei allen Farben die beste ist, ausser bei Weiss, wo RGB besser funktioniert. Dies lässt sich dadurch erklären, dass Weiss im HSV-Farbraum dann entsteht, wenn die Saturation 0 ist. Wenn etwas also komplett weiss ist, dann bedeutet dies, dass der Hue-Wert nicht definierbar ist. Das Programm erkennt somit einen zufälligen Farbton, wenn es ein weisses Feld erkennen muss. Das Pro-

gramm hat somit keine richtige Möglichkeit, die weissen Felder nur anhand des Farbtons zuzuordnen, weshalb es dort zu überdurchschnittlich vielen Fehlern kommt.
Weil die Methode 5 generell die beste ist, wird von nun an ausschliesslich diese verwendet.

2.2 Weisse und schwarze Felder aussortieren

2.2.1 Problem

Wenn nur Hue zur Unterscheidung der Farben verwendet wird, kommt es bei weissen Feldern, die tiefe Saturation-Werte haben, zu Problemen. Dort sind die Hue-Werte fast nicht mehr erkennbar, und die Felder werden folglich zufälligen Farben zugeordnet. Das gleiche gilt auch für schwarze Felder mit zu tiefen Value-Werten. Diese Felder müssen also auf eine andere Weise bestimmt werden.

2.2.2 Lösung

Dieses Problem kann behoben werden, indem man die weissen und die (falls vorhandenen) schwarzen Felder schon am Anfang aussortiert. Dazu werden alle Felder durchgegangen und es wird geschaut, ob sie einen festgelegten tiefen Saturation- oder Value-Wert unterschreiten (vgl. Listing 3).

```
1 if avgcol[side][tile][1] < 50:
2   white_question += 1 # Variable, die weisse Felder zaehlt
3 if avgcol[side][tile][2] < 50:
4   black_question += 1 # Variable, die schwarze Felder zaehlt
```

Listing 3: Test, ob Feld eine tiefe Saturation oder einen tiefen Value hat.

Danach wird geschaut, ob exakt neun Felder einen tiefen Value-Wert besitzen (vgl. Listing 4). Ist dies der Fall, werden die Value-Werte der Mitten in den Zeilen 2 bis 5 verglichen, und die Mitte mit dem tiefsten Wert wird als schwarz festgelegt. Das gleiche Verfahren wird dann noch für das Ermitteln der acht schwarzen Seiten angewendet (nicht im Listing abgebildet). Falls es nicht exakt neun Felder mit einem tiefen Value Wert hat, aber trotzdem einige Felder als schwarz erkannt worden sind, bricht das Programm in Zeile 7 ab und gibt einen Error aus, da der Würfel dann nicht verlässlich erkannt werden kann.

```
1  if black_question == 9:
2    for middle in range(6):
3      if avgcol[middle][8][2] < lowvaluemid: # Vergleich, ob Value
       am tiefsten ist
4        lowvaluemid = avgcol[middle][8][2]
5        blackmidposition = middle
6  elif black_question != 0: # Programmabbruchbedingung
7    sys.exit("ERROR!")
```

Listing 4: Methode, um die schwarze Mitte zu ermitteln.

Nachdem die schwarzen Felder aussortiert worden sind, wird analog das gleiche Verfahren für die weissen, d.h. die Felder mit einer tiefen Saturation angewendet. Dabei muss noch beachtet werden, dass diejenigen Felder, bei denen sowohl die Saturation als auch der Value sehr tief sind, als schwarz erscheinen. Somit sind diese schon aussortiert worden und dürfen nicht mehr beachtet werden.

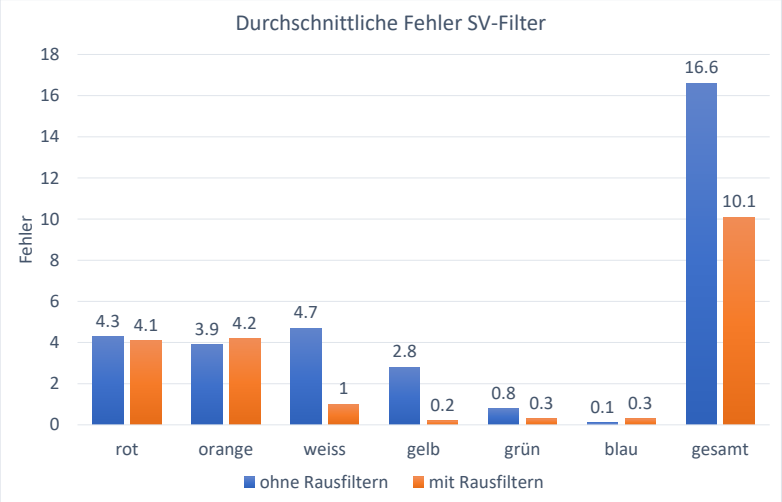

Abbildung 7: Durchschnittliche Anzahl Fehler mit und ohne Anwenden eines Filters für die weissen und schwarzen Felder.

Durch das Anwenden dieses Filters sinkt die durchschnittliche Fehlerzahl von 16.6 auf 10.1 Fehler. Diese Verbesserung kommt hauptsächlich wegen den weissen und gelben Feldern zustande. Von den weissen Feldern wird neu nur noch eins statt 4.7 falsch erkannt, und bei den gelben sind es statt 2.8 noch 0.2 Fehler.

2.3 Korrekte Anzahl Felder pro Farbe

Da bei einem Rubik's Cube bekannt ist, dass jede Farbe exakt neunmal vorkommt, weiss man, dass jeder Mitte exakt acht Felder zugeordnet werden müssen. Ist dies nach der Zuordnung nicht der Fall, lässt sich daraus schliessen, dass der Würfel noch nicht korrekt erkannt ist. In diesem Fall kann man nun noch einen Algorithmus einfügen, der dafür sorgt, dass jede Farbe exakt neunmal vorkommt. So erhöht sich die Chance, dass der Würfel fehlerfrei erkannt wird. Dafür wurden zwei verschiedene Methoden ausprobiert. Die Erste wurde auch in den bisherigen Vergleichen jeweils angewendet.

2.3.1 Nähe zu anderer Mitte

Die erste Methode funktioniert folgendermassen. Falls einer Mitte zu viele Seiten zugeordnet wurden, werden alle diese Seiten mit den anderen Mitten, die zu wenig Seiten haben, verglichen, um zu schauen, welche der Seiten am ehesten zu einer anderen Mitte passt.
Dies wurde im Code wie folgt umgesetzt. In den Zeilen 9 bis 12 wird dafür gesorgt, dass die Abstände der betroffenen Seiten zu den anderen Mitten miteinander verglichen werden, um herauszufinden, welche Seite umgeordnet werden muss. Dafür werden alle Abstände in den Zeilen 13 und 14 auf zwei Bedingungen getestet. Die Erste ist, dass die Mitte weniger als acht ihr zugeordnete Seiten haben muss. Die Zweite ist, dass der Abstand kleiner sein muss, wie der bisher kleinste gespeicherte Abstand. Treffen beide Bedingungen zu, werden in den Zeilen 15 bis 18 der neue Abstand, die Farbe der Mitte und die Position der Seite gespeichert. Nachdem alle Abstände durchgelaufen sind, wird der gespeicherten Seite die neue Farbe zugewiesen, und die Anzahl Felder, die den Mitten zugeordnet sind, werden angepasst.

```
1  for color in range(6):
2    while tilenum[color] > 9:
3      sidepos = 0
4      tilepos = 0
5      colpos = 0
6      counter = 0
7      small_dist = 99999999
8      breaker = 0
9      for side in range(6):
10       for tile in range(8):
11         if tilecol[cubeside][tile] == colors[color]:
12           for diffcol in range(6):
13             if tilenum[diffcol] < 9 and distances[cubeside][
14               tile][
15               diffcol] < small_dist: # Abstand wird verglichen
16               small_dist = distances[cubeside][tile][diffcol]
```

```
17                  sidepos = side # Feld wird gespeichert
18                  tilepos = tile
19                  colpos = diffcol # Farbe wird gespeichert
20      tilecol[sidepos][tilepos] = colors[colpos] # Farbe wird
        geaendert
21      tilenum[color] -= 1 # Anzahl Felder wird angepasst
```

Listing 5: Korrekturmethode, bei der die Seiten, die am nächsten bei einer anderen Mitte sind, umgeordnet werden.

2.3.2 Distanz zu eigener Mitte

Der Aufbau der zweiten Methode ist derselbe wie der der ersten. Auch hier werden die Seiten der Mitten, denen zu viele Seiten zugeordnet wurden, miteinander verglichen. Jedoch werden hier die Abstände zu der aktuell zugeordneten Mitte statt die Abstände zu den anderen Mitten angeschaut. Somit werden bei dieser Methode die Felder, die am weitesten von der eigenen Mitte entfernt sind, umgeordnet.
Im Code wurde es folgendermassen umgesetzt. In den Zeilen 7 bis 9 werden alle betroffenen Seiten durchlaufen. In der Zeile 10 wird geschaut, ob der Abstand zur eigenen Mitte grösser ist, wie der bisher grösste Abstand der anderen Seiten. Ist dies der Fall, werden in den Zeilen 11 bis 13 die Seite und der Abstand gespeichert. Wenn alle Seiten durch sind, wird diese mit dem grössten Abstand einer neuen Mitte zugeordnet. Dies geschieht in den Zeilen 14 bis 19 mit einem ähnlichen Prinzip wie in Methode 1. schliesslich werden in den Zeilen 20 und 21 noch die Variablen für die Anzahl zugeordnete Seiten angepasst.

```
1  for color in range(6):
2    while tilenum[color] > 9:
3      big_dist = 0
4      small_dist = 99999999
5      sidetpos = 0
6      tilepos = 0
7      for cubeside in range(6):
8        for tile in range(9):
9          if tilecol[side][tile] == colors[color]:
10           if big_dist < distances[cubeside][tile][color]: #
       Vergleich von aktuell groesster Distanz und Distanz von Seite
11             big_dist = distances[cubeside][tile][color]
12             sidetpos = side
13             tilepos = tile
14     for diffcol in range(6):
15       if tilenum[diffcol] < 9:
16         if small_dist > distances[sidetpos][tilepos][diffcol]: #
        Zuordnen zu naechster Mitte
```

```
17                small_dist = distances[sidetpos][tilepos][diffcol]
18                colpos = diffcol
19            tilecol[sidetpos][tilepos] = colors[colpos] # Farbe wird
          geaendert
20            tilenum[colpos] += 1 # Anzahl Felder wird angepasst
21            tilenum[color] -= 1
```

Listing 6: Korrekturmethode, bei der die Felder, die am weitesten von der eigenen Mitte entfernt sind, umgeordnet werden.

2.3.3 Vergleich

Die Methode 2 liefert die besseren Resultate, wie im Diagramm unten zu sehen ist. Der Unterschied kommt jedoch nur bei den roten und orangen Feldern zustande. Bei allen anderen Farben sind die beiden Methoden gleich gut.

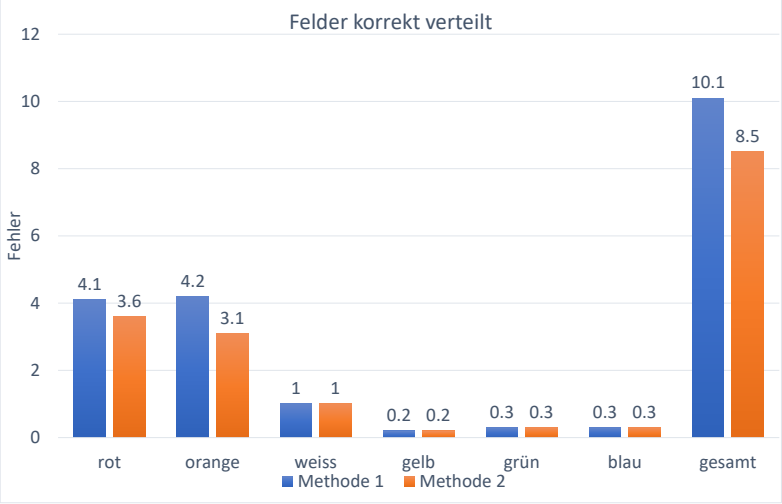

Abbildung 8: Durchschnittliche Anzahl Fehler, mit Algorithmus für die korrekte Anzahl Felder pro Farbe.

3 Fazit

Hier sollen noch einmal die Resultate kurz zusammengefasst werden. Es war immer ziemlich deutlich, welche Methoden besser waren als die anderen. Die besten Ergebnisse werden erzielt, wenn zuerst die weissen und schwarzen Felder herausgefiltert werden, danach die Felder anhand des Hue-Wertes zugeordnet werden, und dann schliesslich bei denjenigen Farben, denen zu viele Felder zugeordnet wurden, die Felder, die am weitesten von der Mitte entfernt sind, noch umgeordnet werden.
Aber auch dann gibt es im Schnitt noch 8.5 Fehler. Dies zeigt, dass es sehr schwierig ist, ein Programm zu erstellen, welches den Rubik's Cube verlässlich und perfekt erkennt. Ein wichtiger Faktor, der dabei eine grosse Rolle spielt, sind die Lichtverhältnisse. Wenn die verschiedenen Seiten auch nur schon einen kleinen Unterschied in der Beleuchtung aufweisen, kann es zu Fehlern kommen. Es wird zwar versucht, diese Fehler zu minimieren, indem nur Hue als Unterscheidungsmerkmal verwendet wird, aber es entstehen trotzdem noch Fehler. Eine andere grosse Fehlerquelle ist das Unterscheiden von Rot und Orange. Da diese beiden Farben einen sehr ähnlichen Hue-Wert haben, ist es schwierig, sie unterscheiden zu können, und schon minimale Störungen im Bild können zu falschen Resultaten führen.
Es gibt aber noch Ideen, die die Anzahl Fehler möglicherweise weiter reduzieren könnten. Zum Beispiel könnte noch geprüft werden, ob das erkannte Muster überhaupt auf einem Rubik's Cube entstehen kann. Es ist beispielsweise nicht möglich, dass auf einem Eckstein eine Farbe mehrmals vorkommt. Es kann auch noch weiter erforscht werden, wie die Veränderung der Lichtverhältnisse die erkannten Farben verändert. Es wäre sodann auch möglich, einen komplett anderen Ansatz zu versuchen, indem man ein neuronales Netz verwendet. Hier käme einfach das Problem hinzu, dass das Trainieren eines neuronalen Netzes möglicherweise Hunderte oder Tausende von Bildern benötigt, was sehr aufwendig wäre.

4 Literatur

[1] Egon L. van den Broek. *Human-centered content-based image retrieval*. 2005. URL: https://www.researchgate.net/figure/The-RGB-color-space-visualized-as-a-cube_fig3_228719004 (besucht am 26. 11. 2019).

[2] Peter Denning. *Can a Rubik's Cube Teach You Programming?* 2016. URL: https://blog.ubiquity.acm.org/can-a-rubiks-cube-teach-you-programming/ (besucht am 26. 11. 2019).

[3] hozela. *cheap security cameras for sale cctv perth best panoramic bullet*. 2019. URL: http://hozela.com/creative-ways-to-use-window-seats/ (besucht am 29. 11. 2019).

[4] Helmut Karger. *Raspberry Video Camera – Teil 17: Exkurs – Wie Computer Farben sehen*. 2017. URL: https://blog.helmutkarger.de/raspberry-video-camera-teil-17-exkurs-wie-computer-farben-sehen/ (besucht am 26. 11. 2019).

[5] O.V. *Farbraum*. 2013. URL: https://www.itwissen.info/Farbraum-color-space.html (besucht am 26. 11. 2019).