Independent Coursework: Untersuchung unterschiedlicher Algorithmen zur Anordnung von Bildern mit Beibehaltung der Seitenverhältnisse

Lilian Hung

15. Oktober 2018

Inhaltsverzeichnis

Liste der Beispiele

1 Einführung

Auf der Suche nach einem geeigneten Algorithmus, der Bilder ihrer Ähnlichkeit nach anordnet, dabei das Seitenverhältnis der Bilder ("aspect ratio") beibehält und den Zwischenraum der Bilder minimiert, werden im Folgenden drei verschiedene Paper untersucht, indem die gegebenen Algorithmen implementiert und daraufhin ausgewertet werden, wie gut sie gewisse Eingaben darstellen. Zwei weitere Paper, welche versuchen ähnliche Probleme zur Bildanodnung zu lösen, wurden ebenfalls untersucht und in Abschnitt 5 zusammengefasst.

1.1 Versuchsaufbau

Als Eingabe wird eine Menge von Rechtecken unterschiedlicher Farbe und Größe verstanden, die als Repräsentation von Bildern gesehen werden. Diese Rechtecke haben eine Sortierung bzw. Position im zwei-dimensionalen Raum, welche zeigt wie ähnlich sich die Rechtecke in Bezug auf eine bestimmte Eigenschaft sind.

In den praktischen Versuchen wurde hauptsächlich auf eine 32×32 Ein-

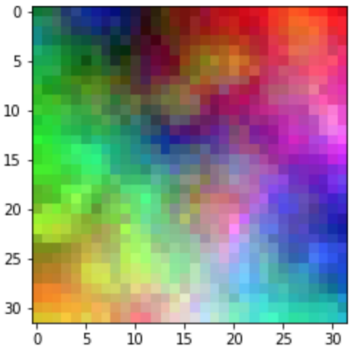

Abbildung 1: 32×32 SOM

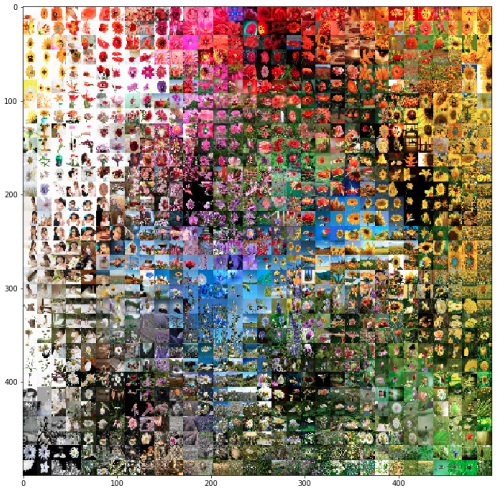

Abbildung 2: 32×32 vorsortierte Bilder von Blumen

heiten/Pixel große Self-Organized-Map (SOM) zurückgegriffen (Siehe Abbildung 1) und somit war das Ähnlichkeitsmaß in diesem Fall die Farbe der Rechtecke. Die SOM wurde zuvor berechnet und aus dem gegebenen Bild wurde dann ein zwei-dimensionaler Array erstellt, der die Farben der Pixel speichert. In derselben Größe (32×32) wurde daraufhin ein Array erstellt, der Rechtecke unterschiedlicher Seitenlängen und entsprechenden, aus dem SOM-Farb-Array zugeordneten, Farben enthält. Der *RWordle*-Algorithmus in Abschnitt 3 verwendet zusätzlich auch die sortierten Bilder von Blumen (siehe Abbildung 2).

1.1.1 Erstellung der Rechtecke

Die *Python*-Implementierung zur Erstellung der Rechtecke (Beispiel 1) wird in der Umsetzung der Algorithmen *RWordle* und *PicWall* verwendet. Die Rechtecke werden als Tupel in einem Array `rects` mit Informationen über die Position, Höhe und Breite und einer Farbe abgelegt.

```
1  rects = []
2  for x in range(0, 32):
3      for y in range(0, 32):
4          rand_width = randint(30,100)
5          rand_height = randint(30,100)
6          all_width = all_width + rand_width
7          all_height = all_height + rand_height
8          r = float(col_arr[x][y][0]/255)
9          g = float(col_arr[x][y][1]/255)
10         b = float(col_arr[x][y][2]/255)
11         rects.append( (x, y, rand_width, rand_height, (r,g,b,
       0.6)) ) #alpha 0.6
```

Beispiel 1: Erstellung der Rechtecke; `col_arr` enthält die Farben aus der SOM

Zur Darstellung der Rechtecke wird die `pyplot`-Funktion in Kombination mit der `patches`-Funktion, beides Funktionen der `mathplotlib`-Bibliothek, verwendet.

Die Implementierung der Generierung der Datenpunkte für *Nmap* wurde in *JavaScript* mit Hilfe des Paketes `get-pixels` [4] umgesetzt.

2 *Nmap*

Das Paper *Nmap: A Novel Neighborhood Preservation Space-filling Algorithm*[1] behandelt die Erstellung einer *Tree-Map* unter Beachtung der Position und des Gewichtes der gegebenen Datenpunkte. Die Beachtung der Position und damit das Ziel Ähnlichkeitsbeziehungen zwischen den Elementen beizubehalten ist ein entscheidender Punkt, in dem sich *Nmap* von anderen Tree-Maps unterscheidet. *Nmap* steht für *Neighborhood Treemap*. Der allgemeine Ansatz, der in *Nmap* verwendet wird, ist der *Slice-And-Scale*-Ansatz, bei welchem die gegebene Fläche geteilt (*slice*) und dann die entstandenen zwei Flächen skaliert (*scale*) werden.

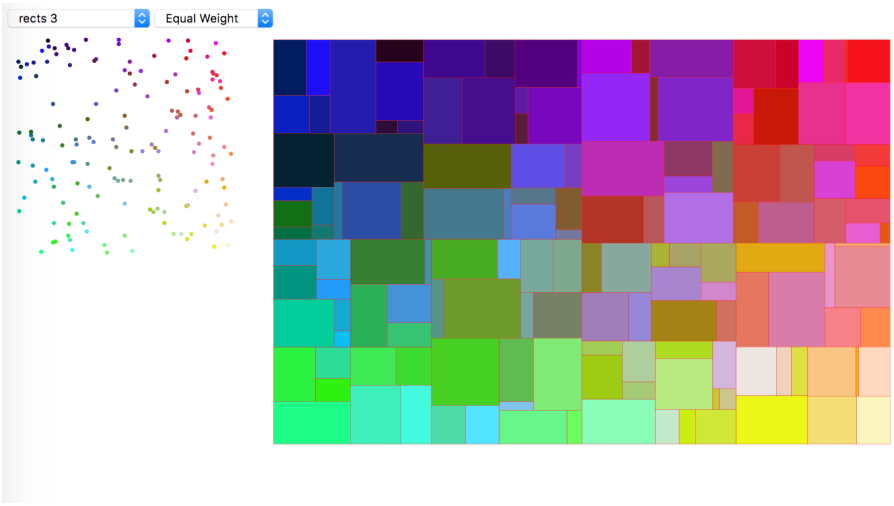

Abbildung 3: Erste Versuche mit *Nmap*. Die Datenpunkte wurden mit randomisierter Position und Größen/Gewichten generiert.

2.1 Algorithmus

Im Paper [1] werden zwei Formen von *Nmap* beschrieben – *Nmap-AC* ("Alternate Cut") und *Nmap-EW* ("Equal Weight"). In *Nmap-AC* wird das zu füllende Rechteck abwechselnd horizontal und vertikal geteilt, die erste Teilrichtung ist in einem Fall, wenn das Rechteck höher als breit ist, horizontal und im anderen Fall vertikal. Bei horizontaler Teilung werden die Datenpunkte an ihrem Median nach Sortierung anhand der y-Koordinate

geteilt, bei vertikaler Teilung wird der Median nach Sortierung anhand der x-Koordinate ermittelt. Dann werden die beiden neuen Rechtecke nach den Gewichtsummen der enthaltenen Datenpunkte skaliert. Dieser Schritt wird rekursiv wiederholt bis ein Rechteck nur noch einen Datenpunkt enthält. In *Nmap-EW* werden die Datenpunkte nicht am Median geteilt, sondern so, dass beide Teilmengen der Datenpunkte das selbe Gewicht enthalten.

2.2 Implementierung und Auswertung

Es gibt bereits zugängliche Implementierungen von *Nmap* in *Java* und *JavaScript*. Die Implementierung in *JavaScript* visualisiert die Daten mit *D3* und lässt sich auf *Github* finden und klonen (siehe [2]). Die geklonte und angepasste Implementierung befindet sich hier [3]. Um die Implementierung am gegebenen Anwendungsfall zu testen, müssen Datenmengen in Form von CSV-Dateien generiert werden. Außerdem musste der *D3*-Code zur Visualisierung angepasst werden, damit Datenpunkte *mit Farb-Informationen* entsprechend angezeigt werden können. Die Generierung der Daten ist in *JavaScript* implementiert und kann im Terminal mit *node* ausgeführt werden. Für die ersten Tests wurden nur Daten mit unterschiedlichen Farben und Größen bzw. Gewichten generiert, welche nach ihrer rot- und grün-Komponente im zwei-dimensionalen Raum angeordnet wurden. Später wurden mit der gegebenen 32×32 SOM, Datenpunkte in der durch die SOM vorgegebenen zwei-dimensionalen Sortierung und mit unterschiedlichen Gewichten generiert.

2.2.1 Anmerkung zur Generierung der Datenpunkte

Bei einer Generierung muss sowohl bei den Gewichten der Datenpunkte als auch bei deren Position darauf geachtet werden, dass diese alle leicht unterschiedlich sind. Bei den Positionen ist damit der Abstand zwischen den Datenpunkten gemeint. Sollten diese identisch sein, kann es zu Lücken in der späteren Visualisierung durch *Nmap* kommen. In der Dokumentation von nmap.js wird zwar darauf hingewiesen, dass eine Visualisierung von Elementen mit gleichem Gewicht diesen Darstellungsfehler hervorruft, jedoch konnte ich den Fehler auch bei gleichen Abständen zwischen Datenpunkten feststellen. Der Fehler lässt sich beim Gewicht und bei den Koordinaten der Daten durch eine Addition einer kleinen Zufallszahl beheben (`[...]x: 50 + Math.random(), y: 50 + Math.random(), weight: 1000 + Math.random()`

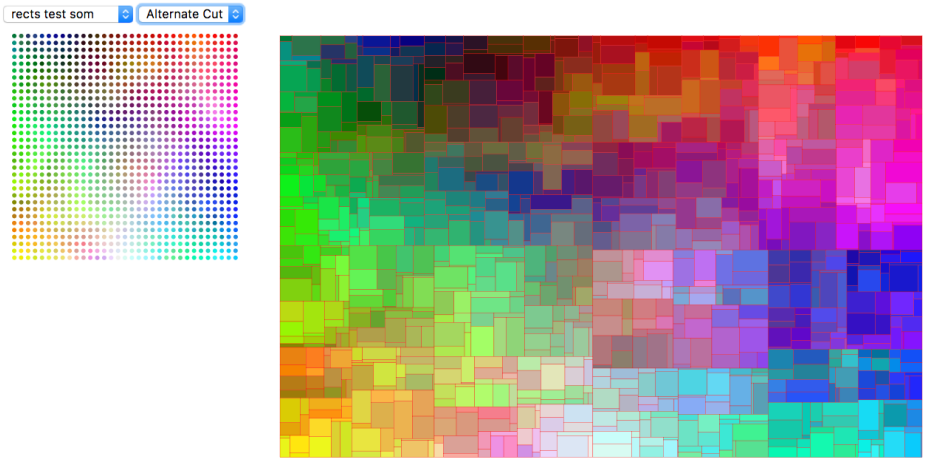

Abbildung 4: Visualisierung von *Nmap-AC* mit Datenpunkten, die aus der gegebenen SOM mit randomisierten Größen/Gewichten generiert wurden.

`[...]`).

2.2.2 Auswertung

Da *Nmap* eine lückenfreie Aufteilung einer vorgegebenen Fläche zurückgibt und dabei die Nachbarschaften der visualisierten Elemente gut aufrecht erhält, scheint es für den Anwendungsfall vorsortierte Bilder bzw. Rechtecke zu visualisieren geeignet zu sein. Damit die Eingabemenge für *Nmap* passend ist, braucht man für jedes Element, in diesem Fall ein Bild oder ein Rechteck, eine Position in Form einer x und y-Koordinate sowie ein Gewicht. Das Gewicht kann als die Höhe mal die Breite eines Rechtecks oder Bildes definiert werden. An dieser Definition kann bereits erkannt werden, dass in der späteren Visualisierung zwar eine Fläche dem Bild entsprechend dargestellt wird, aber das ursprüngliche Seitenverhältnis nicht garantiert erhalten bleibt. Somit fällt ein essenzieller Teil des zu lösenden Problems weg und es muss für *Nmap* gefolgert werden, dass es sich, ohne größere Veränderungen, nicht als Lösung eignet.

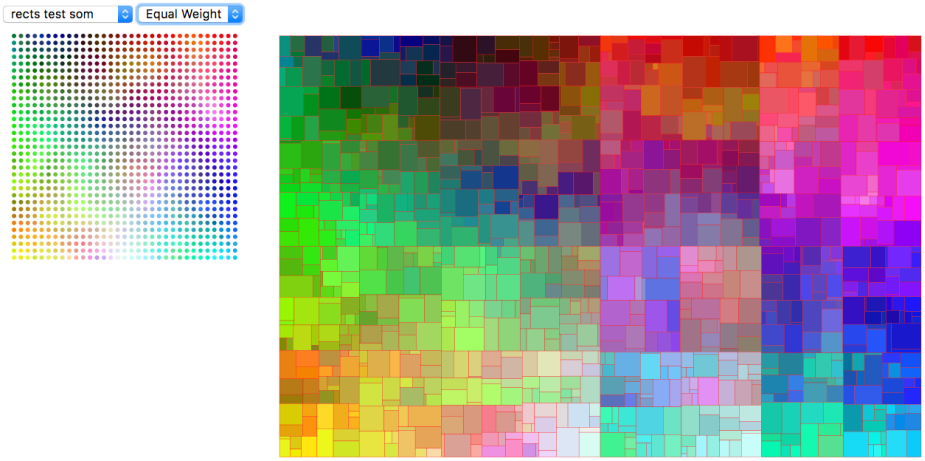

Abbildung 5: Visualisierung von *Nmap-EW* mit Datenpunkten, die aus der gegebenen SOM mit randomisierten Größen/Gewichten generiert wurden.

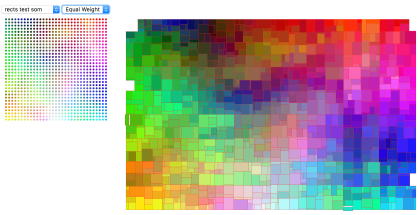

Abbildung 6: Fehlerhafte Visualisierung von *Nmap-EW*, da die Datenpunkte keinen leicht randomisierten Abstand zueinander haben.

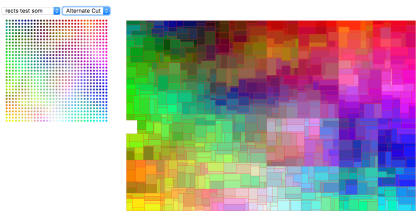

Abbildung 7: Fehlerhafte Visualisierung von *Nmap-AC*, da die Datenpunkte keinen leicht randomisierten Abstand zueinander haben.

3 *RWordle*

Das Problem, das in *Rolled-out Wordles: A Heuristic Method for Overlap Removal of 2D Data Representatives* [5] zu lösen versucht wird, ist die Aufhebung von möglichen Überlagerungen von flächigen Objekten (in diesem Fall Rechtecke oder wie im Paper genannt "Label") durch neue Positionierung. Der ursprüngliche Anwendungsfall ist die Aufhebung von Überlagerungen, die oftmals beim Mapping von höher-dimensionalen Daten auf eine zweidimensionale Fläche entstehen.

3.1 Algorithmus

Es gibt zwei Versionen von *RWordle*: In der einen Version wird die Datenmenge genommen und entlang einer linearen Achse (beispielsweise der y-Achse) sortiert und die Elemente der Datenmenge dieser Reihenfolge nach neu positioniert (*RWordle-L*). Das erste Element behält seine Position und die folgenden Elemente werden um die bereits positionierten Elemente herum positioniert. Im Genaueren wird die neue Position ausgehend von der ursprünglichen Position des Rechtecks bestimmt. Die Suche erfolgt spiral- bzw. kreisförmig um die ursprüngliche Position herum (siehe Abbildung 8). Die andere Version sortiert die Datenmenge nach einer konzentrischen Sortierung (*RWordle-C*). Damit ist gemeint, dass die Elemente nach ihrem Abstand zum Mittelpunkt der gesamten Datenmenge sortiert werden. Die Elemente werden ebenfalls entsprechend ihrer Reihenfolge neu positioniert.

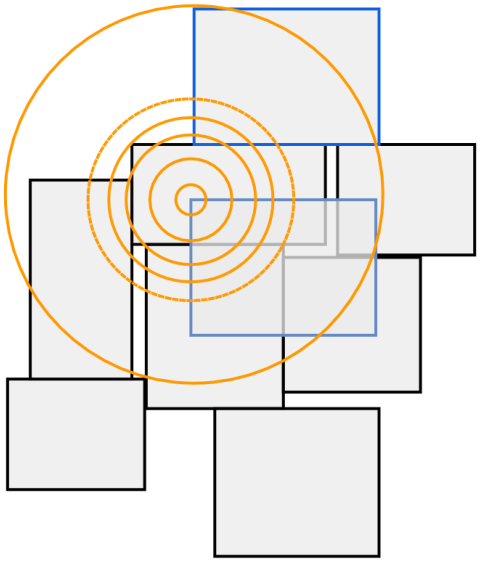

Abbildung 8: Suche nach einer neuen Position für ein Rechteck ohne Überlagerungen auf einem größer werdendem Kreis mit Mittelpunkt an der linken oberen Ecke der ursprünglichen Position des Rechtecks

3.2 Implementierung und Auswertung

Die Implementierung und graphische Auswertungen von *RWordle* wurden in *Python* umgesetzt. Gegeben war zur Untersuchung eine vorberechnete 32×32 SOM (Self-Sorting-Map), wie in der Einführung kurz erwähnt, um festzustellen, ob sich der Algorithmus für unseren Anwendungsfall eignet. Da in den Versuchen mit der Anordnung von Rechtecken, wie auch in Abbildungen 10 und 11 zu sehen, Ergebnisse tatsächlich zum Anwendungsfall passen, wurde auch noch eine Implementierung, die mit Bildern als Eingabemenge arbeitet, umgesetzt. Die Bilder als Eingabemenge waren, ähnlich wie die Farb-SOM, als im zwei-dimensionalen Raum vorsortierte 32×32 Einheiten gegeben (siehe Abbildung 2). Daraus mussten noch Teilbilder generiert werden, die zueinander unterschiedliche Seitenverhältnisse haben.

3.2.1 Erste Umsetzung mit Rechtecken

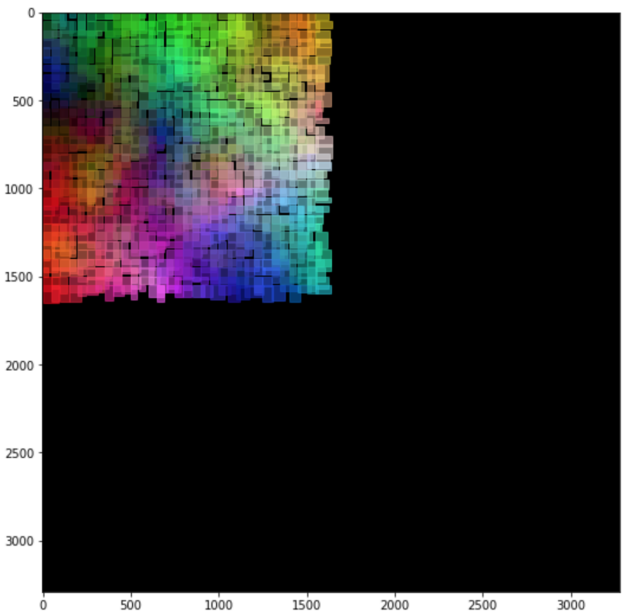

Abbildung 9: Die Positionen und der Rechtecke in ihrer ursprünglichen Anordnung. Die Rechtecke sind mit einer Transparenz von 0.6 Abgebildet, um die Überlagerungen mit den anderen Rechtecken sichtbar zu machen.

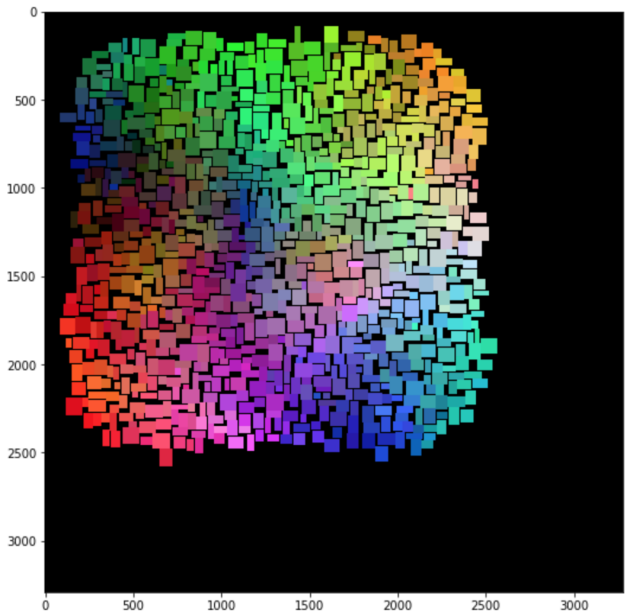

Abbildung 10: Die Anordnung der Rechtecke nachdem *RWordle* mit einer konzentrischen Sortierung ausgeführt wurde

In der ersten Umsetzung wurde die SOM (siehe Abbildung 1), aus welcher ein Array mit Rechtecken zufälliger Größe erstellt wurde, verwendet. Die Rechtecke wurden außerdem leicht umpositioniert. Die implizite Position der Rechtecke kann als deren Position im Array verstanden werden. Da dies bei den Rechtecken dazu führt, dass eine sehr starke Überlappung auftritt, waren die ersten Ergebnisse von *RWordle* nur mäßig gut. Die Ähnlichkeitsbeziehungen bleiben bei *RWordle* relativ gut erhalten, jedoch kann die ursprüngliche Form, welche ein Quadrat ist, nicht gut erhalten werden. Es entsteht eine kreisförmige Anordnung der Rechtecke. Diese Anordnung ergibt sich nicht aus der konzentrischen oder linearen Vorsortierung, sondern aus der starken Überlagerung. Da die Rechtecke ausgehend von ihrer originalen Position einen neuen Platz ohne Überlagerung auf einem kreis- bzw. spiralförmigen Weg suchen, liegt der neue Platz immer möglichst nah an den bereits positionierten Rechtecken. Somit entsteht am Ende eine kreisförmige Anordnung, wenn die Rechtecke zuvor alle übereinander lagen und daher alle Rechtecke – außer dem ersten Rechteck – eine neue und insbesondere weit von ihrer ursprünglichen Position entfernten Position finden müssen. Hat man eine etwas

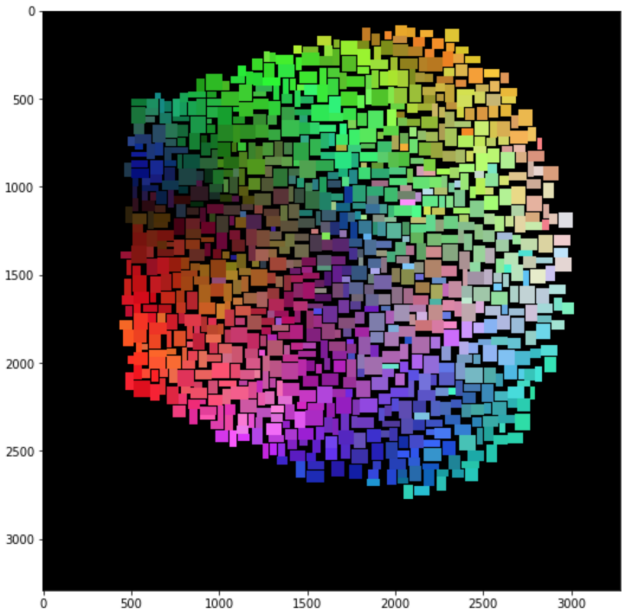

Abbildung 11: Die Anordnung der Rechtecke nachdem *RWordle* mit einer linearen Sortierung ausgeführt wurde

aufgefächerte Anordnung der Rechtecke als Ausgangspunkt, kann die Form besser erhalten werden, da die ersten Rechtecke in der Anordnungsreihenfolge in ihrer neuen Position weniger von ihrer ursprünglichen Position abweichen. Dementsprechend weichen auch die Positionen der später bis zuletzt angeordneten Rechtecke weniger von ihrer ursprünglichen Position ab. Die Umpositionierung der Rechtecke von ihren zu stark überlappenden Positionen erfolgt über eine Skalierung der x- und y-Koordinate um einen gewissen Faktor. Wird dieser zu groß gewählt, tritt keine Überlagerung mehr auf, aber der Platz zwischen den Rechtecken wird auch nicht minimiert. Daher ist eine so starke Überlagerung gesucht, bei deren Aufhebung durch *RWordle* eine Anordnung entsteht, die der ursprünglichen Form noch ähnlich ist.

3.2.2 Erweiterung um Umsetzung mit Bildern

Algorithmisch musste zur Anordnung mit Bildern nur wenig am Code verändert werden. Die Bilder mussten jedoch im Gegensatz zur Generierung der Rechtecke anders beschaffen werden. Dazu ist, wie in der Einleitung kurz erwähnt, ein Bild mit 32×32 sortierten Bildern gegeben. Diese sortierten Bilder sind in

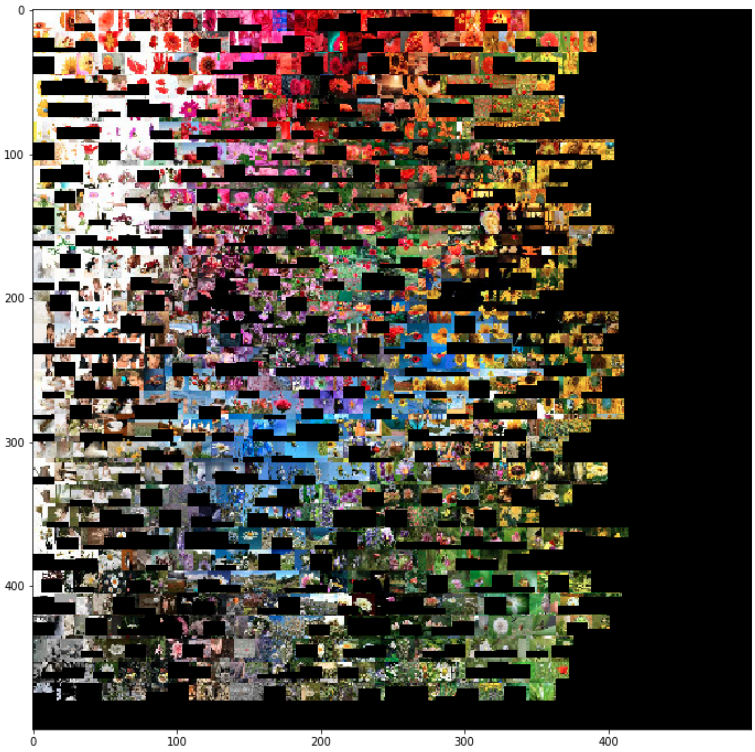

Abbildung 12: Einfache Anordnung der Bilder nebeneinander.

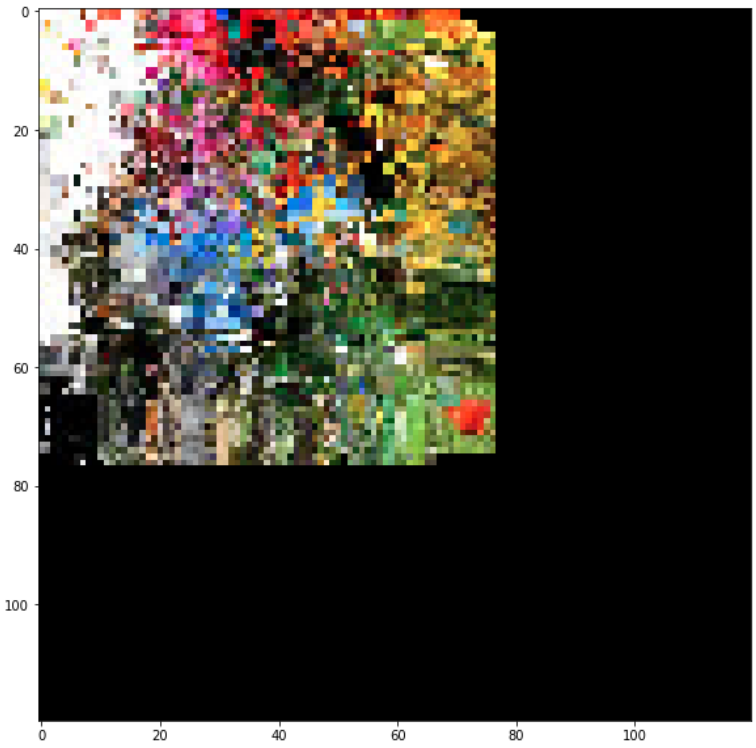

Abbildung 13: Starke Überlappung der Bilder vor der Anwendung von *RWordle*.

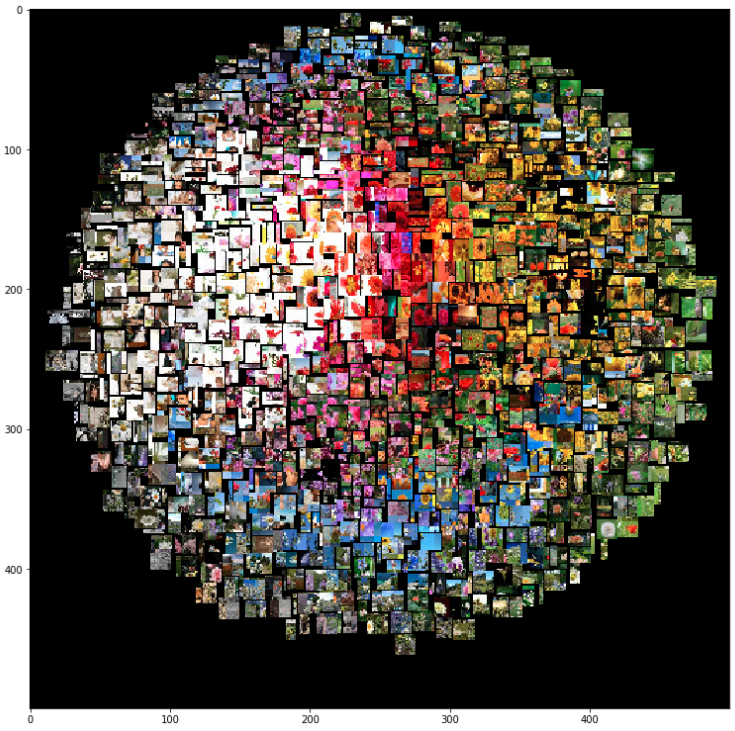

Abbildung 14: Anordnung der Bilder durch *RWordle* mit einer konzentrischen Sortierung. Die Bilder haben sich zuvor stark überlappt und die Form der Bilderanordnung konnte nicht erhalten werden.

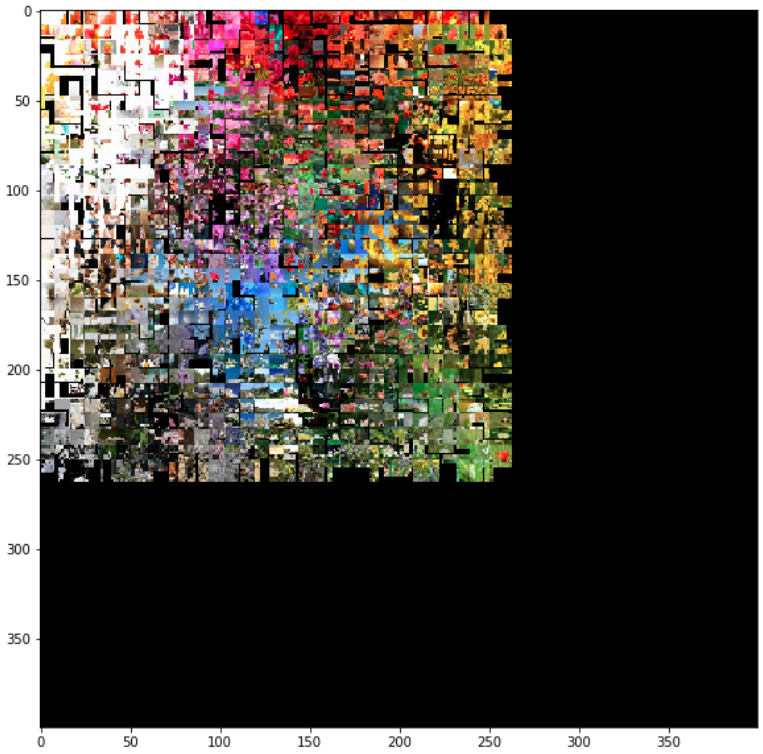

Abbildung 15: Weniger starke Überlappung der Bilder vor der Anwendung von *RWordle*.

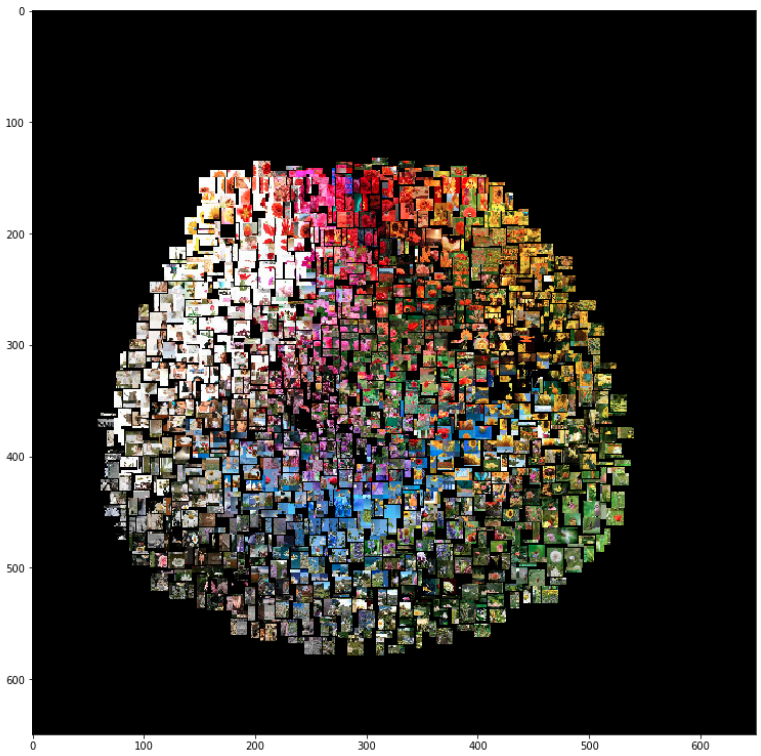

Abbildung 16: Anordnung der Bilder durch *RWordle* mit einer konzentrischen Sortierung. Die Bilder haben sich zuvor weniger stark überlappt (Abbildung 15) und die Form der Bilderanordnung wurde etwas besser erhalten.

der gegebenen Form alle quadratisch. Für eine Eingabemenge, deren Bilder unterschiedliche Seitenverhältnisse haben, wird aus den quadratischen Bildern ein Teilbild ausgeschnitten. Für die Teilbilder wird ein Seitenverhältnis zwischen 1 bis 5 für die Höhe und 1 bis 5 für die Breite zufällig erzeugt. Diese Bilder sind zur Veranschaulichung einer nicht besonders gut geeigneten, jedoch einfach zu erzeugenden Anordnung, in Abbildung 12 nebeneinander platziert worden.

3.2.3 Die `updatePosition`-Funktion

Um die Bilder oder Rechtecke neu zu positionieren wird die `updatePosition`-Funktion aufgerufen. Diese hat als Eingabeparameter das neu zu positionierende Rechteck bzw. Bild und einen Array, in dem die bereits positionierten Bilder enthalten sind. Um wie beschrieben eine neue Position kreis- bzw. spiralförmig um die ursprüngliche Position herum verlaufend zu finden, werden die Variablen `angle` bzw. `angle_stepsize` und `radius` bzw. `radius_stepsize` verwendet. Diese definieren, wie ihre Namen besagen, einen Winkel und einen Radius. Der Radius beschreibt letztlich den Abstand zur ursprünglichen Position. Diese Größen können beliebig klein oder groß skaliert werden. Es sollte darauf geachtet werden, dass der initiale Wert für `radius` und auch der Wert für `max_radius` von der Größe, Anzahl und Verteilung der Bilder bzw. Rechtecke und der Größe der gegebenen Fläche abhängt und sinnvoll gesetzt werden müssen. `angle_stepsize` und `radius_stepsize` geben die Granularität vor, in der nach einer neuen Position gesucht wird. Je feiner diese ist, desto länger braucht man zur Positionierung der Bilder/Rechtecke.

```
1  def updatePosition(rect, layoutedRects):
2      angle = 0
3      angle_stepsize = pi/36
4      radius = 5
5      radius_stepsize = 5
6      max_radius = radius_stepsize * 1000
7      overlap = True
8
9      while(overlap and radius <= max_radius): #iterate through
       each angle for radius increased by radius_stepsize until
      max_radius
10         while(angle < 2*pi):
11             tmpRect = ()
```

```
12              tmpRect = tmpRect + ( (rect[0] + radius * cos(
    angle)), )
13              tmpRect = tmpRect + ( (rect[1] + radius * sin(
    angle)), )
14              for a in range(2, len(rect)):
15                  tmpRect = tmpRect + ( rect[a], )
16
17              overlapping =  overlaps(tmpRect, layoutedRects
    [0])
18              for i in range(1, len(layoutedRects)):
19                  #if any layouted rect still overlaps with new
     position of tmpRect
20                  overlapping = overlapping or overlaps(tmpRect
    , layoutedRects[i])
21
22              if(not overlapping):
23                  overlap = False
24                  return tmpRect
25
26              #otherwise update angle
27              angle = angle + angle_stepsize
28
29          #reset angle
30          angle = 0
31          radius = radius + radius_stepsize
```

Beispiel 2: `updatePosition`-Funktion zur Positionierung der Rechtecke für *RWordle*

3.2.4 Auswertung

Im Vergleich zur Darstellung der Bilder/Rechtecke in aufgefächerter Form nebeneinander liefert *RWordle* ein Ergebnis, welches in die gewünschte Richtung geht. Da *RWordle* jedoch ein Algorithmus zur Entfernung von Überlagerungen in einer Menge von Objekten ist, hat er nur Auswirkungen auf Eingabemengen, in denen sich die Elemente tatsächlich überlagern. Er dient nicht in erster Linie der Anordnung von Bildern/Rechtecken in kompakter Form, es ist ein Resultat der Beseitigung der Überlagerung von Elementen und der Versuch die Anordnung und vorherige Position in Relationen zu den anderen Elementen so gut wie möglich beizubehalten. Wie in den Ergebnissen (siehe Abbildungen 10, 11, 14, 16) zu erkennen, werden die Nachbarschaften der Elemente insgesamt gut beibehalten.

Allerdings kann man vor allem bei kleineren Rechtecken/Bildern beobachten, dass diese die Nähe zu ihren ursprünglichen Nachbarn nicht so gut aufrecht erhalten können, da sie in Lücken hineinpassen, in die größere Rechtecke/-Bilder nicht hineingepasst haben. Unter anderem bleibt ihre Position auf Grund von Umplatzierungen der Elemente um sie herum unangetastet oder sie finden bei der Suche nach einer neuen Position freie Stellen, in die die ursprünglichen Nachbarn nicht hineingepasst haben und somit weiter entfernt ihre neue Position finden mussten.

4 *PicWall*

In *PicWall: Photo collage on-the-fly* [6] wird ein Algorithmus zur automatischen Erstellung von Fotokollagen auf unterschiedlich großen Flächen vorgestellt. Für ein gegebenes Seitenverhältnis einer rechteckigen Fläche soll eine solche Aufteilung gefunden werden, in der die gegebenen Bilder in ihren originalen Seitenverhältnissen angezeigt werden und dabei möglichst wenig Zwischenraum freilassen.

4.1 Algorithmus

Um eine passende Anordnung für eine Menge von Bilder mit unterschiedlichen Seitenverhältnissen in einer recheckigen Fläche zu finden, werden zwei grundlegende Funktionen benötigt. Zum einen die Berechnung von dem Seitenverhältnis der rechteckigen Fläche, welche sich aus den Seitenverhältnissen der Bilder ergibt, und zum anderen das Finden einer passenden Position in der Aufteilung der rechteckigen Fläche.

Zur Umsetzung wird ein Divide-And-Conquer-Ansatz verfolgt, indem das Rechteck rekursiv randomisiert vertikal oder horizontal aufgeteilt wird bis genügend Rechtecke entstehen, um alle gegebenen Bilder anzuzeigen. Diese Aufteilung wird in einer binären Baumstruktur widergespiegelt, wobei die Blätter die einzelnen Unterbereiche repräsentieren und die inneren Knoten die Information darüber beinhalten, ob eine vertikale oder horizontale Teilung der Fläche gegeben ist.

Nach der Unterteilung der gegebenen Fläche wird für jedes Blatt ein Bild gesucht, dessen Seitenverhältnis am besten zu dem Seitenverhältnis des Unterbereichs passt. In der Visualisierung wird das Bild so skaliert, dass es in den Unterbereich passt, jedoch ohne die Seitenverhältnisse zu verändern

oder das Bild zu beschneiden.

Anfangs wird im Paper beschrieben wie erst das rekursive Berechnen des gesamten Seitenverhältnisses erfolgt (siehe *Algorithmus 1* in [6]). Darauffolgend wird erläutert wie die Berechnung der Positionen der linken oberen Ecke der Unterbereiche abläuft (siehe *Algorithmus 2* in [6]). Da dies noch keine optimale Lösung liefert und auch keine nutzerdefinierte Flächengröße sinnvoll aufgeteilt werden kann, wird auf Verbesserungen und Erweiterungen eingegangen. Zum einen lässt sich eine bessere Aufteilung ohne großen Aufwand finden, wenn für einen inneren Knoten mit zwei Blättern als Kinder die Bilder nicht einzeln pro Blatt bestimmt werden, sondern zwei Bilder gefunden werden, die gut in das Seitenverhältnis des inneren Knotens passen (siehe *Algorithmus 3* in [6]). Zum anderen wird eine Aufbesserung der zufällig generierten horizontalen und vertikalen Aufteilung durch eine Iteration über den Baum, bei der für jeden Knoten untersucht wird, ob statt einer horizontalen oder vertikalen Aufteilung entsprechend eine vertikale oder horizontale Aufteilung besser zum erwarteten Seitenverhältnis passt. Sollte das der Fall sein, wird das Label im Knoten umgeschrieben und die Aufteilung der Fläche somit verbessert (siehe *Algorithmus 4* in [6]).

4.2 Implementierung und Auswertung

In der vorliegenden Implementierung wird ein Baum zur Aufteilung der Fläche generiert indem ein Knoten mit der `Node`-Klasse als Wurzel `root` des Baumes erstellt wird. Die Parameter sind im Folgenden aufgelistet.

- Ein Seitenverhältnis `aspectRatio`
- Die Anzahl der Bilder bzw. der Blätter `numOfLeaves`
- Das Elternelement `parent` wird bei der Wurzel auf `None` gesetzt und enthält ansonsten den Elternknoten
- Die konkrete Höhe und Breite des Unterbereichs, den der Knoten repräsentiert (`width`, `height`)
- Die Position der linken oberen Ecke des Unterbereichs `position` einen Array mit den Rechtecken
- Wenn das Element ein Blatt ist, ein Bild `image`, ansonsten ist dies `None`

Ein Knoten besitzt alle Informationen, die er benötigt, um rekursiv Knoten für sein linkes und rechtes Kind zu erstellen. Falls der Knoten ein Blatt ist, wird ein passendes Bild mit der Funktion `findBestImg` gesucht und festgelegt. Im zweiten Rekursionsanker (*base case*), wenn beide Kinder Blätter sind, wird die Funktion `findImagePair` benötigt. *Diese hat in der momentanen Implementierung noch einen Fehler, weswegen versucht wird die vorhandenen Rechtecke/Bilder mehrfach zu verwenden.* Im Paper ([6]) konnte ich keine genauere Beschreibung finden, wie für einen Knoten mit zwei Blättern als Kinder bestimmt wird, welche Teilungsrichtung, horizontal oder vertikal, er hat. Der vierte Algorithmus, im Paper ([6]) *Fast Tree Adjustment* genannt, wurde bisher noch nicht umgesetzt. Dieser sollte allerdings das Gesamtergebnis sehr verbessern.

4.2.1 Auswertung

In den Abbildungen 17 und 18 sieht man das Ergebnis, wobei nur ein zufälliger Baum für 32×32 Rechtecke berechnet wird. Für jedes Blatt wurde versucht ein im Seitenverhältinis möglichst passendes Rechteck zu finden. Da keine Anpassung von vertikaler bzw. horizontaler Teilung eines Unterbereichs vorgenommen wird, sind viele Bereiche ungeeignet. Bei einer Erstellung einer Aufteilung mit nur 3×3 Rechecken, ist diese etwas besser (siehe Abbildungen 19 und 20). Um eine aussagekräftigere Auswertung für *PicWall* zu erhalten, müssen auf jeden Fall noch die fehlenden Funktionen implementiert und deren Fehler korrigiert werden. Leider hat dies nicht mehr in den zeitlichen Rahmen dieser Arbeit gepasst.

Eine letzte Anmerkung zu *PicWall* ist, dass trotz der fehlenden Anpassung der Aufteilung und der Erweiterung zur Festlegung von zwei Rechtecken gleichzeitig, sichtbar ist, dass in *PicWall* keine zuvor zwischen Bildern/Rechtecken bestehenden Nachbarschafts- bzw. Ähnlichkeitsbeziehungen erhalten werden, da diese Information zur Positionierung der Bilder nicht verwendet wird.

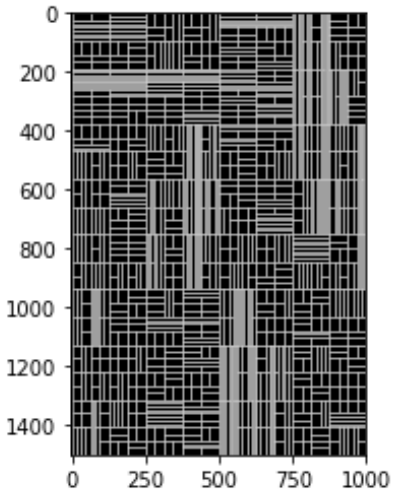

Abbildung 17: Visualisierung der randomisierten Aufteilung in Unterbereiche. Verwendung von allen 32×32 Rechtecken.

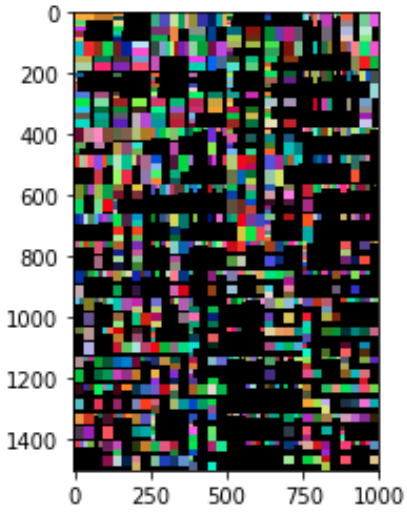

Abbildung 18: Die Anordnung der Rechtecke/Bilder nachdem *PicWall* ausgeführt wurde. Verwendung von allen 32×32 Rechtecken.

5 Weitere untersuchte Algorithmen

5.1 *Tree-based Visualization and Optimization for Image Collection*

In *Tree-based Visualization and Optimization for Image Collection* [7] werden die Bilder gestreckt und gestaucht, um eine möglichst lückenfreie Anordnung der Bilder zu erhalten. Die Ähnlichkeitsbeziehungen zwischen Bildern bleiben gut erhalten, da die Bilder einer zuvor erstellten Baumstruktur nach positioniert werden. Es wird beschrieben, wie diese Baumstruktur zu erstellen ist, aber es kann auch ein bereits erstellter Baum als Eingabe verwendet und visualisiert werden.

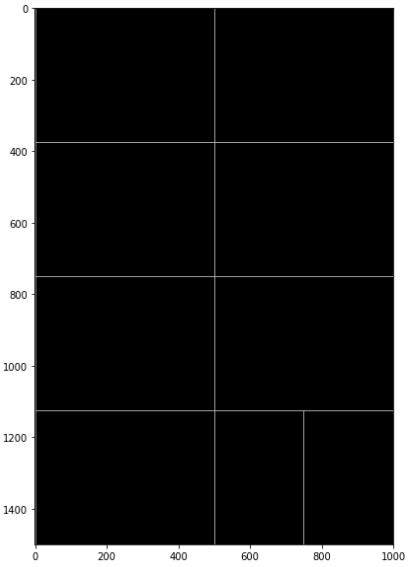

Abbildung 19: Visualisierung der randomisierten Aufteilung in Unterbereiche. Verwendung von 3×3 Rechtecken.

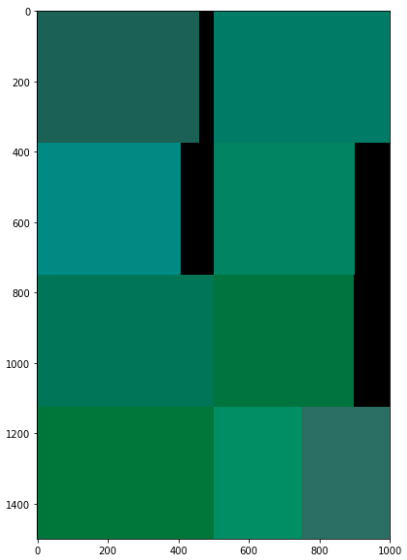

Abbildung 20: Die Anordnung der Rechtecke/Bilder nachdem *PicWall* ausgeführt wurde. Verwendung von 3×3 Rechtecken.

5.2 *Photo Layout with a Fast Evaluation Method and Genetic Algorithm*

In *Photo Layout with a Fast Evaluation Method and Genetic Algorithm* [8] wird eine Methode vorgestellt, mit der man Collagen erstellen kann. Dabei wird das Seitenverhältnis beachtet und ebenfalls versucht die gegebenen Bilder möglichst lückenfrei anzuordnen. Ebenfalls wird erwähnt, dass es dem Nutzer möglich sein soll, ein besonderes Bild auszuwählen, welches größer als die anderen dargestellt werden soll. Die Anordnung der Bilder basiert auf einer binären Baumstruktur, die der von *PicWall* sehr ähnlich ist. Allerdings wird ein geeigneter Baum basierend auf der *Genetic Algorithm*-Methode gefunden, indem eine Menge unterschiedlicher binärer Bäume generiert wird und darunter dann der am besten geeignete ausgewählt wird. Eine Möglichkeit Ähnlichkeitsbeziehungen zwischen den Bildern durch räumliche Nähe darzustellen wird nicht beschrieben.

Literatur

[1] F. S. L. G. Duarte, F. Sikansi, F. M. Fatore, S. G. Fadel and F. V. Paulovich, *Nmap: A Novel Neighborhood Preservation Space-filling Algorithm*, in IEEE Transactions on Visualization and Computer Graphics, vol. 20, no. 12, pp. 2063-2071, 31 Dec. 2014. DOI: 10.1109/TVCG.2014.2346276, URL: `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6876012&isnumber=6935054`

[2] Javascript Implementierung von Nmap *nmap.js*, `https://github.com/sebastian-meier/nmap.js/`, 20.09.2018.

[3] Geklonte und angepasste Implementierung von Nmap *nmap.js*, `https://github.com/revialim/nmap.js`, 09.10.2018.

[4] NPM-Paket `get-pixels`, `https://www.npmjs.com/package/get-pixels`, 09.10.2018.

[5] H. Strobelt, M. Spicker, A. Stoffel, D. Keim, and O. Deussen. 2012. *Rolled-out Wordles: A Heuristic Method for Overlap Removal of 2D Data Representatives.* Comput. Graph. Forum 31, 3pt3 (June 2012), 1135-1144. DOI=`http://dx.doi.org/10.1111/j.1467-8659.2012.03106.x`

[6] Z. Wu and K. Aizawa, *PicWall: Photo collage on-the-fly*, 2013 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference, Kaohsiung, 2013, pp. 1-10. DOI: 10.1109/APSIPA.2013.6694305, URL: `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6694305&isnumber=6694103`

[7] X. Han, C. Zhang, W. Lin, M. Xu, B. Sheng and T. Mei, *Tree-Based Visualization and Optimization for Image Collection*, in IEEE Transactions on Cybernetics, vol. 46, no. 6, pp. 1286-1300, June 2016. DOI: 10.1109/TCYB.2015.2448236, URL: `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7155544&isnumber=7466429`

[8] J. Fan, *Photo Layout with a Fast Evaluation Method and Genetic Algorithm*, 2012 IEEE International Conference on Multimedia and Expo Workshops, Melbourne, VIC, 2012, pp. 308-313. DOI: 10.1109/ICMEW.2012.59, URL: `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6266273&isnumber=6266221`