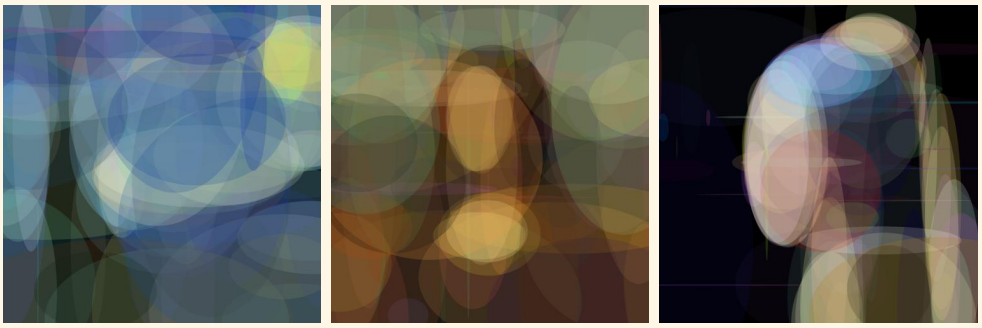
Kunst durch Evolution



Evolutionäre Algorithmen in der Bildkompression

von Aleksandrs Beloseikins

Humboldt-Gymnasium Potsdam
Heinrich-Mann-Allee 103
14473 Potsdam

Schuljahr 2021/22

Seminararbeit
im Seminarkurs Bionik

- Kunst durch Evolution - Evolutionäre Algorithmen in der Bildkompression

von
Aleksandrs Beloseikins

Betreuende Lehrkräfte:
Frau Peter & Frau Brandt

Abgabetermin: 08.10.2021

Inhaltsverzeichnis

1 Einleitung **1**

2 Biologisches Vorbild **1**
2.1 Definition Evolution 1
2.2 Natürliche Selektion 2
2.3 Rekombination 3
2.4 Mutation 3

3 Evolutionäre Algorithmen **4**
3.1 Modellbildung und -übertragung 4
3.2 Initialisierung 4
3.3 Selektion 5
3.4 Rekombination 5
3.5 Mutation 5

4 Computergrafik **6**
4.1 RGB- und RGBA-Farbraum 6
4.2 Farbdifferenzen 8
4.3 Rastergrafik 8
4.4 Vektorgrafik 9

5 Implementation **9**
5.1 Zielsetzung und Hypothese 9
5.2 Algorithmus 10
5.2.1 Programmaufbau 10
5.2.2 Initialisierung 11
5.2.3 Fitnessfunktion und Selektion 11
5.2.4 Mutation 12
5.2.5 Kompression und Dekompression 12
5.3 Auswertung 13
5.3.1 Durchführung 13
5.3.2 Informationsverlust 14
5.3.3 Kompression 16
5.3.4 Laufzeit 17
5.3.5 Vergleich mit JPEG 19

6 Fazit **20**
6.1 Verbesserungsansätze 20
6.2 Ausblick 21

7 Literatur und Abbildungsverzeichnis **22**

A Quellcode **25**

B Ergebnisse **32**

1 Einleitung

In der heutigen digitalisierten Welt müssen Millionen von Dateien in Sekundenbruchteilen hochgeladen, gespeichert und verschickt werden. Dabei sollten diese möglichst kompakt gespeichert sein, um Ladezeiten zu verringern und Speicherplatz zu sparen. Für diesen Zweck wurden bereits mehrere Kompressionsalgorithmen entwickelt. In dieser Arbeit soll erforscht werden, ob evolutionäre Algorithmen auch hierfür verwendet werden können.
Evolutionäre Algorithmen sind inspiriert von der Evolution von Lebewesen in der Natur und gehören somit zum Forschungsgebiet der Bionik. Zum ersten Mal wurden sie in den 1950ern und 60ern erforscht [1, S.2] und fanden bereits Anwendungen in zahlreichen Optimierungsproblemen. Dazu gehören das „Travelling Salesman"-Problem, das Optimieren von Zeitplänen, das frühe Erkennen von Krankheiten, aber auch das Rekonstruiren von Bildern, was zeigt, dass mit ihnen auch in der Computergrafik bereits Erfolge erzielt wurden [2]–[5].
Um nun die Frage zu klären, ob evolutionäre Algorithmen auch im Gebiet der Bildkompression eingesetzt werden könnten, wird im Verlauf der Arbeit das biologische Vorbild evolutionärer Algorithmen und ihr allgemeiner Ablauf beschrieben, die Art und Weise, wie Bilder von Computern gespeichert werden erläutert und ein selbst entwickelter evolutionärer Algorithmus nach bestimmten Kriterien ausgewertet. Zum Schluss wird ein Fazit zur Leitfrage gezogen und ein Ausblick gegeben.

2 Biologisches Vorbild

2.1 Definition Evolution

Um evolutionäre Algorithmen in ihrer Gänze nachvollziehen zu können, muss zunächst sein biologisches Vorbild, die Evolution, beschrieben werden. Eine mögliche Definition der Evolution ist die „stammesgeschichtliche Entwicklung der Organismenarten" [6, S.298]. Diese Entwicklung von Organismen über Generationen entsteht durch das Zusammenspiel dreier Evolutionsfaktoren: der natürlichen Selektion, der Rekombination und der Mutation [6, S.311]. Im Folgenden sollen diese Evolutionsfaktoren genauer beschrieben werden. Es sei dabei gesagt, dass eine stark vereinfachte Version der Evolution erläutert wird, die für das Verständnis evolutionärer Algorithmen genügt.

2.2 Natürliche Selektion

Als natürliche Selektion - auch natürliche Auslese genannt - wird der Prozess bezeichnet, bei dem an ihre Umstände angepasstere Organismen, also Organismen mit einer höheren Fitness, eine größere Überlebenschance haben, somit mehr Nachkommen zeugen können und ihr Genotyp, also die Gesamtheit ihrer Erbanlagen, dadurch im Genpool prävalenter vertreten ist [6, S.280]. Die natürliche Selektion ist deswegen der richtungsgebende Evolutionsfaktor, da durch sie spätere Generationen immer angepasster an ihre Umgebung werden. Die Faktoren, die darüber bestimmen, wie gut ein Organismus an seine Umgebung angepasst ist, werden als Selektionsfaktoren (oder Auslesefaktoren) bezeichnet und sind unterteilt in biotische und abiotische Umweltfaktoren. Biotische Umweltfaktoren bezeichnen die Faktoren, die durch wechselseitige Beziehungen zwischen Lebewesen entstehen. Einen solchen Faktor stellen Fressfeinde dar. Abiotische Umweltfaktoren hingegen bezeichnen die unbelebten Auswirkungen auf Organismen, wie beispielsweise das Klima. [6, S.310f]. Es wird außerdem in drei Selektionstypen unterschieden: die stabilisierende, transformierende und disruptive Selektion. Alle drei beschreiben Weisen, wie sich bestimmte Merkmale von Organismen ändern können. In der stabilisierenden Selektion wird für den derzeitigen Medianwert eines Merkmals und gegen die Extremwerte selektiert. Bei der transformierenden Selektion kommt es zu einer Selektion eines Wertes des Merkmals, welcher vom Median abweicht. Es kommt dadurch zu einer Verschiebung des Medians. Schließlich ist die disruptiven Selektion das Gegenstück zur stabilisierenden Selektion, da es dort zu einer Selektion für die Extrema und gegen den Mittelwert kommt. Dabei entstehen 2 neue Hochpunkte für je ein Extremum des Merkmals, die sich beide vom ursprünglichen unterscheiden [7]. Die Auswirkungen der Selektionstypen auf die Population sind in Abbildung 1 dargestellt.

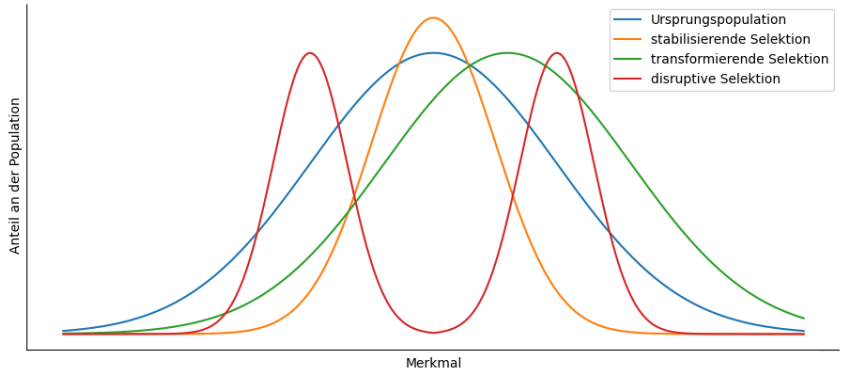


Abbildung 1: grafische Darstellung der 3 Selektionstypen

2.3 Rekombination

Die Rekombination bezeichnet die Neuverteilung von genetischem Material während der sexuellen Fortpflanzung. Ein wichtiger Prozess, bei dem Rekombination stattfindet, ist die Meiose. In dieser entstehen aus einer diploiden Mutterzelle vier haploide Tochterzellen, da es während der Meiose zwei Mal zu einer Zellteilung kommt. Während der Meiose kommt es zur zufälligen Kombination aus mütterlichen und väterlichen Chromosomen, also Neukombination ganzer Chromosome. Ein weiterer, für evolutionäre Algorithmen bedeutsamerer, Prozess, der auch während der Meiose stattfindet, ist das Überkreuzen von Chromosomen. Dies wird auch „crossing over“ genannt und führt dazu, dass Chromosomteile zwischen 2 homologen Chromosomen, also Chromosomen der selben Größe und Form, ausgetauscht werden. Dadurch werden einzelne Gene zwischen den beiden Chromosomen getauscht und so neu kombiniert. Diese beiden Prozesse werden auch inter- und intrachromosomale Rekombination genannt. [6, S.267, 271f]

2.4 Mutation

Mutationen beschreiben die Prozesse, die zur Änderung des Erbgutes bei Organismen führen. Es existieren Genommutationen, Chromosomenmutationen und Genmutationen. Genommutation beschreiben die Vergrößerung oder Verkleinerung der Chromosomenanzahl, während bei Chromosomenmutationen eine Änderung der Struktur von Chromosomen stattfinden. Besonders interessant für evolutionäre Algorithmen sind jedoch die Genmutationen, bei welchen es zu Veränderungen in einzelnen Genen kommt. Dies kann unter Anderem durch das Austauschen einer Base durch eine andere geschehen. Durch die Genmutation kommt es zur Bildung von Allelen, also unterschiedlichen Versionen desselben Gens [6, S.269, 288].
Die Mutation sorgt gemeinsam mit der Rekombination für eine genetische Variation. Beide können zu Veränderung des Phänotyps, also der Gesamtheit aller Merkmale eines Organismus, führen. Diese Änderungen können sowohl positiv, als auch negativ sein. Die natürliche Selektion selektiert dann die am besten angepassten Organismen [6, S.311].

3 Evolutionäre Algorithmen

3.1 Modellbildung und -übertragung

Die Evolution scheint deshalb interessant für Optimierungsprobleme, da der richtungsgebende Faktor die Selektion, also die Bewertung der Fitness eines Organismus ist. Dies ist insofern von Bedeutung, dass es für viele Probleme einfacher ist, die Qualität einer Lösung zu bewerten, als einen Algorithmus für die direkte Lösung zu entwickeln.
Wie in der Bionik typisch, wird anfangs ein abstrahiertes biologisches Modell entwickelt, um ein technisches Modell zu bilden, welches dann implementiert werden kann [8, S.27ff]. Vereinfacht man den Prozess der Evolution, so erhält man die zwei wichtigsten Eigenschaften, die zur „Optimierung“ der Organismen beitragen - die zufällige Variation des Erbgutes durch Mutation und Rekombination und die Selektion der fittesten Individuen. Sollen nun mehrere Variablen, die beispielsweise die Eigenschaften einer Antenne beschreiben, optimiert werden, können diese Variablen als Genom, die Gesamtheit aller Gene, betrachtet werden. Damit stellt jedes Individuum der Population eine mögliche Lösung des Optimierungsproblems dar. Über die Generationen werden diese Lösungen durch Rekombination und Mutation zufällig verändert, woraufhin die besten selektiert werden. Der allgemeine Ablauf lässt sich somit folgendermaßen beschreiben [1, S.11f]:

1. Initialisierung einer zufälligen Population
2. Selektion der Besten der Population
3. Rekombination ihrer Genome und Erstellung einer neuen Population
4. Mutation der Genome der neuen Population
5. beliebig häufige Wiederholung der Schritte 2-4
6. Rückgabe des Genoms des besten Individuums der letzten Population

Die einzelnen Schritte werden im Weiteren genauer erklärt.

3.2 Initialisierung

Am Anfang des evolutionären Algorithmus wird eine neue Population mit p Individuen generiert, wobei p für die Populationsgröße steht. Dabei wird das gesamte Genom jedes Individuums randomisiert. Das Genom besteht aus einer Reihe von Variablen, die für die Lösung des Optimierungsproblems relevant sind und die nun auf zufällige Werte gesetzt werden. Wie genau das geschieht, ist von Problem zu Problem unterschiedlich. Im Kontext dieser Arbeit stellen

die Variablen Zahlenwerte dar, womit bei der Initialisierung die Variablen auf Zufallszahlen in einem gewählten Intervall gesetzt werden. Daraufhin wird die Anzahl der Generationen g festgelegt, die bestimmt, wie viele Generationen durchgeführt werden, wobei eine Generation aus Selektion, Rekombination und Mutation besteht.

3.3 Selektion

Am Anfang jeder Generation wird eine Fitnessfunktion auf alle Individuen der Generation angewandt. Die Fitnessfunktion stellt die Gesamtheit aller Fitnessfaktoren dar und bestimmt, wie nah die jeweilige Lösung am Optimum ist. Bei der Optimierung einer Brücke würde sie beispielsweise das Gewicht und die Stabilität bewerten [8, S.93]. Die Ausgabe der Fitness-Funktion ist ein beliebiger Zahlenwert und wird als Fitness des Individuums gespeichert. Nachdem die Fitness jedes Individuums bestimmt wurde, beginnt die Selektion. Dabei werden in Abhängigkeit der Fitness Individuen für die Rekombination selektiert. Das kann bedeuten, dass die besten $N\%$ (wobei N arbiträr gewählt werden kann) selektiert werden oder dass die Wahrscheinlichkeiten so gewichtet werden, dass die besten der Population eine höhere Chance und die schlechtesten der Population eine kleinere Chance haben, selektiert zu werden [9, S.30f]. Anzumerken beim ersteren ist, dass bei höheren N eine größere Variation in der Population vorliegt und bei kleineren N der Algorithmus zielgerichteter eine Lösung optimiert - dafür aber andere, potentiell bessere Lösungen, übersehen kann. Die Variation an verschiedenen Lösungen wird als Suchbreite bezeichnet.

3.4 Rekombination

Nachdem 2 Individuen aus der Population selektiert wurden, findet die Rekombination (auch Crossover genannt) statt. Dabei wird durch das gesamte Genom der beiden Elternteile iteriert und zufällig Allele von einem der Individuen in ein neues Genom kopiert. Nachdem dadurch ein komplett neues Genom gebildet wurde, wird es einem neuen Individuum zugeteilt und dieses in eine neue Population gespeichert. Sobald durch diese Schritte eine neuer Population der Größe p entstanden ist, wird die ursprüngliche Population durch die neue ersetzt[1, S.10].

3.5 Mutation

Da durch die Rekombination keine neuen Allele dazukommen, sondern nur alte neu kombiniert werden, sorgt die Mutation für die Entstehung dieser. Je-

des Gen wird mit einer Wahrscheinlichkeit P_M zufällig mutiert. Wie schon bei der Initialisierung kann das für verschiedene Probleme unterschiedliche Operationen bedeuten. Zahlen werden bei der Mutation um einen zufälligen Wert, der einem gewählte Intervall liegt, verändert - dieser Zufallswert wird also auf den Originalwert addiert. Nachdem die Mutation aller Individuen abgeschlossen ist, fängt die nächste Generation an, bis die Generationsanzahl g erreicht wird, wonach das beste Individuum der letzten Generation zurückgegeben wird [1, S.10].

4 Computergrafik

4.1 RGB- und RGBA-Farbraum

Der RGB-Farbraum ist neben den HSL- bzw. HSV-Farbraum der am häufigsten verwendete Farbraum. „RGB“ steht dabei für „Red-Green-Blue“, also Rot-Grün-Blau, welche die Primärfarben dieses Farbraums sind. Der RGB-Farbraum ist ein additiver Farbraum, was bedeutet, dass jede Farbe im Farbraum als Summe dieser 3 Farben gebildet werden kann. Hat man 3 Lampen in den Farben Rot, Grün und Blau und mit einstellbarer Helligkeit, so kann man durch Überlagerungen ihres Lichtes und Änderungen der Helligkeit der einzelnen Lampen alle Farben des Farbraums erzeugen [10]. Stellt man den RGB-Farbraum als dreidimensionales Koordinatensystem dar (Abbildung 2), wobei die Vektoren $\vec{r} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$, $\vec{g} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$, $\vec{b} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ die Basis sind, dann lässt sich eine Farbe als Punkt in diesem Koordinatensystem darstellen. In der Abbildung gezeigt sind die Farbpunkte an den R-G-, G-B- und R-B-Ebenen. In der Wirklichkeit gibt es noch mehr Farben im Inneren des „Würfels“, die man aber nicht alle gleichzeitig anzeigen kann. Die in der Abbildung angezeigten Farben, die man durch Überlagerung 2 der 3 RGB-Farben erhält, werden auch Sekundärfarben genannt[10].

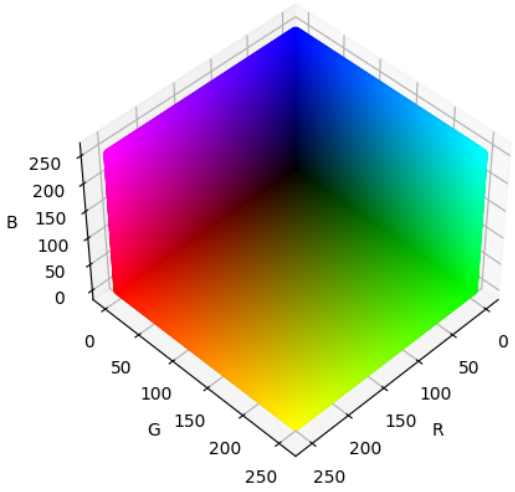


Abbildung 2: RGB-Farbraum als dreidimensionales Koordinatensystem

Der Ortsvektor jedes Punktes lässt sich durch Linearkombination der Vektoren $\vec{r}, \vec{g}, \vec{b}$ bilden. Als Skalare können dafür Ganzzahlen im Intervall $[0; 255]$ gewählt werden. Das liegt daran, dass so für jeden der drei RGB-Werte nur 1 Byte Speicher verwendet werden kann [10]. Ein Byte besteht aus 8 Bits, die jeweils den Wert 0 oder 1 annehmen können. Damit kann ein Byte $2^8 = 256$ verschiedene Werte speichern und somit alle Ganzzahlen zwischen 0 und 255 darstellen. Insgesamt beinhaltet der RGB-Farbraum $256 \cdot 256 \cdot 256 = 16777216$ verschiedene Farben, die alle durch 3 Bytes eindeutig beschrieben sind. Da jeder Punkt für eine Farbe steht, kann man Farben als Tupel der Form (R,G,B) beschreiben. Eine Eigenschaft, die später in der Auswertung relevant wird, ist dass bei R=G=B Grauwerte entstehen, die bei höheren R,G,B heller und bei kleineren R,G,B dunkler werden.
Optional kann im RGBA-Format noch ein vierter Parameter - die Transparenz α - eingestellt werden. Diese ändert die Farbe nicht, bestimmt aber wie sich Farben beim Überlagern verhalten [11]. Je höher der α-Wert ist, desto stärker sieht man die Farbe und desto schwächer erkennt man Farben im Hintergrund - die Transparenz ist also geringer. Bei kleineren α-Werten wird die Farbe selbst immer transparenter. Der α-Wert liegt zwischen 0 (komplett transparent) und 1 (nicht transparent). Beim Überlagern der Farben A und B, wobei A über B gelegt wird, gilt für die resultierende Farbe C [11]:

$$\begin{aligned} C &= \frac{A \cdot \alpha_A + B \cdot \alpha_B(1 - \alpha_A)}{\alpha_C} \\ \alpha_C &= \alpha_A + \alpha_B(1 - \alpha_A) \end{aligned} \tag{1}$$

Dabei stehen A, B, C für die RGB-Werte der Farben und α_X für den α-Wert der Farbe X. Man kann an der Gleichung erkennen, dass bei höheren α_A die Farbe B hinter A einen immer kleiner werdenden Einfluss besitzt und bei $\alpha_A = 1$ komplett unsichtbar wird. Bei kleinen α_A sieht man den genau umgekehrten Effekt und bei $\alpha_A = 0$ ist A komplett transparent und man sieht nur noch die Farbe B dahinter.

4.2 Farbdifferenzen

Die geometrische Darstellung des RGB-Farbraums hat insofern einen Vorteil, dass die Berechnung der Differenz zweier RGB-Werte intuitiver ist. Da jede Farbe ein Punkt im Raum ist, kann man für die Differenz zweier Farbwerte die euklidische Distanz der Punkte bzw. den Betrag des Differenzenvektors der Ortsvektoren berechnen. Damit ergibt sich der Unterschied zweier Farben im RGB-Format als:

$$\left| \begin{pmatrix} R_1 \\ G_1 \\ B_1 \end{pmatrix} - \begin{pmatrix} R_2 \\ G_2 \\ B_2 \end{pmatrix} \right| = \sqrt{(R_1 - R_2)^2 + (G_1 - G_2)^2 + (B_1 - B_2)^2} \quad (2)$$

4.3 Rastergrafik

Eine Rastergrafik, auch Pixelgrafik genannt, beschreibt die Darstellung eines Bildes in einer Matrix der Form $W \times H$, wobei W die Breite und H die Höhe des Bildes beschreibt. Jedes Element der Matrix wird als Pixel bezeichnet. Jedem Pixel wird eine Farbe zugeordnet. Da zuvor erläutert wurde, dass Farben im RGB-Format durch 3 Zahlenwerte beschrieben werden, besteht jedes Element der Matrix aus einem 3-Tupel, welches die Farbe des Pixel beschreibt. In Implementationen wird das häufig mit dreidimensionalen Arrays gelöst, wobei die RGB-Werte die dritte Dimension einnehmen. Besitzt das Bild eine Transparenz, muss zusätzlich der α-Wert gespeichert werden, womit jedes Element ein 4-Tupel beschreibt. Da dieser Fall für den weiteren Verlauf der Arbeit jedoch keine Bedeutung hat, da nur mit nichttransparenten Rastergrafiken gearbeitet wird, wird er nicht weiter beachtet.
Sollen zwei Bilder, die als Rastergrafik gespeichert sind, verglichen werden, kann der MSE(Mean Squared Error)-Algorithmus genutzt werden. Dieser kommt vor allem in der Statistik vor, kann aber zum Vergleich zweier Bilder verwendet werden. Wie bereits beschrieben, berechnet sich der Unterschied zweier Farbwerte als die euklidische Distanz ihrer Punkte im Farbraum. Der MSE berechnet die Summe der quadratischen Differenzen der RGB-Werte geteilt durch die Anzahl der Pixel. Damit berechnet er die gemittelte Distanz der

Farbwerte mit dem Unterschied, dass die Wurzel ausgelassen wird, womit es die quadrierte Distanz wird [12]. Das liegt daran, dass die Wurzelfunktion aufwendig zu berechnen ist und sie auf jeden einzelnen Pixel angewandt werden müsste, worunter die Laufzeit leiden würde. Seien A und B zwei Bilder, die verglichen werden, so gilt:

$$MSE(A, B) = \frac{1}{W \cdot H} \cdot \sum_{j=1}^{H} [\sum_{i=1}^{W} (A_{ij} - B_{ij})^2] \tag{3}$$

,wobei mit $(A_{ij} - B_{ij})^2$ die komponentenweise quadrierte Differenz der RGB-Werte gemeint ist.

4.4 Vektorgrafik

Rastergrafik hat jedoch einen großen Nachteil, der beim Skalieren der Bilder auftritt. Wird ein Bild mit einer hohen Breite und Höhe auf eine kleinere Breite und Höhe skaliert, dann müssen mehrere Pixel in einen einzelnen zusammengefasst werden, was dazu führt, dass Information aus dem Originalbild verloren geht. Wird ein kleineres Bild größer skaliert, führt das im Umkehrschluss dazu, dass einzelne Pixel zu mehreren Pixeln werden. Da keine Information dazukommt, weisen die neu dazugekommenen Pixeln keinen Informationsgehalt auf und es sieht so aus, als wäre jeder Pixel des Originals vergrößert worden. Das Bild erscheint somit „verpixelt“[10].
Um diesem Problem entgegenzuwirken wurde die Vektorgrafik eingeführt. Bei dieser wird ein Bild nicht als Matrix aus Pixeln gespeichert, sondern als eine Reihe von mathematischen Objekten, wie Ellipsen und Polygonen. Dies hat den Vorteil, dass beim Skalieren die Auflösung gleich hoch bleibt, da statt dem Zusammenfassen bzw. Vervielfachen von Pixeln, einzelne Werte, wie beispielsweise der Radius eines Kreises oder die Position eines Punktes, verändert werden können. Deswegen findet die Vektorgrafik Anwendung bei Schriftarten oder Logos [13]. Ein weit verbreitetes Bildformat, das Vektorgrafiken speichert, ist SVG (Scalable Vector Graphics).

5 Implementation

5.1 Zielsetzung und Hypothese

Um zu überprüfen, ob evolutionäre Algorithmen in der Bildkompression verwendet werden können, wurde ein solcher evolutionärer Algorithmus eigenständig implementiert. Dabei wurde die Programmiersprache Python verwendet, was

zum einen an persönlicher Präferenz, aber auch an der guten Lesbarkeit der Sprache liegt. Genauer wurde ein evolutionärer Algorithmus entwickelt, der ein Bild in eine Menge aus halbtransparenten Ellipsen konvertiert. Anschließend werden diese binär kodiert und in einer Datei gespeichert. Als Inspiration dafür diente ein Blog-Eintrag von Roger Johansson, in welchem er die Mona Lisa in halbtransparenten Polygonen darstellte [14]. Dadurch entstand die Idee, vom Speicherplatz kompaktere Formen als Polygone zu nutzen und zu versuchen, diese möglichst effizient zu speichern. Ein Vorteil dieses Ansatzes neben der Kompression liegt auch darin, dass die resultierenden Ellipsen als Vektorgrafik gespeichert werden könnten, deren Vorteile zuvor erläutert wurden.
Im Anschluss wird der Algorithmus anhand von drei Kriterien bewerten. Das erste Kriterium ist der Informationsverlust, also wie stark das komprimierte Bild vom Original abweicht und worin diesbezüglich die Probleme des Verfahrens liegen. Es wird vermutet, dass es eine relativ starke Abweichung vom Original geben wird, da die Ellipsen vor allem feine Strukturen und kleinere Objekte nicht berücksichtigen werden. Der zweite Aspekte ist die Kompression, also wie wenig Speicherplatz das komprimierte Bild einnehmen wird. Die Kompression hängt vermutlich davon ab, wie viele Ellipsen verwendet werden - die Kompression nimmt höchstwahrscheinlich mit bei größerer Ellipsenanzahl ab. Bei ca. 100 Ellipsen sollte die komprimierte Datei unter 10 Kilobytes einnehmen. Das letzte Kriterium, nach welchem der Algorithmus bewertet wird, ist die Laufzeit und wie sie mit der Anzahl von Ellipsen und Generationen zunimmt. Es wird die Vermutung aufgestellt, dass mit steigender Ellipsen- und Generationsanzahl auch die Laufzeit steigen wird, da der Algorithmus über beide iteriert.

5.2 Algorithmus

5.2.1 Programmaufbau

Die Implementation besteht aus 2 Dateien - *main.py*, welche den eigentlichen evolutionären Algorithmus und die Speicherung dem komprimierten Bildes enthält, und *decoder.py*, welche eine Funktion enthält, die die komprimierte Datei wieder in ein von herkömmlichen Programmen leserliches Bildformat konvertiert. *decoder.py* besteht aus nur einer Funktion, weswegen im Weiteren *main.py* genauer erläutert wird. Da evolutionäre Algorithmen stark von der Natur inspiriert wurden, bietet es sich an, ein Programmierparadigma zu verwenden, welches sich an der echten Welt orientiert. Aus diesem Grund wurde sich für einen objekt-orientierten Ansatz entschieden. Dafür wurden 3 Klassen definiert - `Ellipse`, `Picture` und `Population`. `Ellipse` enthält alle Eigenschaften,

die die Ellipsen definieren: Die x- und y-Koordinaten, die Länge der Halbachsen und die RGB-Werte für ihre Farbe. Es wurde bewusst auf die Rotation verzichtet, da diese der Kompression schadet. Das liegt daran, dass für 360° jeweils 2 Bytes pro Ellipse dazu kommen müssten, da 1 Byte nur bis zu 256 Ganzzahlwerte speichern könnte. Zudem wurde der α-Wert statisch auf 20% gestellt, um wiederum Speicherplatz zu sparen und da Versuche mit einem veränderbaren α-Wert schlechtere Ergebnisse erzeugt haben. Die Klasse `Ellipse` beinhaltet außerdem die Funktionen `mutate()`, welche eine Ellipse zufällig mutiert und `random()`, welche die Parameter der Ellipse randomisiert. Diese ist wichtig für die Initialsierung. Die Klasse `Picture` repräsentiert das Individuum der Population. Sie enthält einen Array aus Ellipsen `ellipses`, die zusammen das Bild ergeben. Zudem besitzt sie auch die Funktionen `mutate()`, `random()`, `fitness_func()` und `output()`. `mutate()` und `random()` mutieren bzw. randomisieren alle dazugehörigen Ellipsen aus `ellipses`, während `fitness_func()` die Fitness-Funktion darstellt, die später genauer beleuchtet wird, und `output()` die Ellipsen als Bild exportiert. Diese Funktion wird größtenteils zum Debuggen verwendet, da das endgültig komprimierte Bild erst am Programmende gespeichert wird. Abschließend stellt die Klasse `Population` die Population dar. Sie enthält die Liste `pictures` mit allen Bildern bzw. `Picture`-Objekten, die Funktion `generation()`, die die Selektion, Rekombination und Mutation durchführt und die Funktion `run()`, welche `generation()` so oft aufruft, wie die Generationsanzahl eingestellt wurde. Der gesamte Quellcode befindet sich im Anhang A.

5.2.2 Initialisierung

Beim Ausführen des Programms wird ein ausgewähltes Bild aus dem Verzeichnis geladen und in einem dreidimensionalen Array gespeichert, wobei auch die Dimensionen des Bildes (Höhe und Breite) als Array gespeichert werden. Dann wird eine neue Population aus `Picture`-Objekten mit jeweils randomisierten `Ellipse`-Objekten generiert, wofür die `random()`-Funktionen genutzt werden. Die Größe der Population und die Anzahl der Ellipsen sind dabei frei einstellbar. Anschließend wird die Funktion `run()` der `Population`-Klasse mit einem Parameter aufgerufen, der die Generationsanzahl bestimmt, womit der eigentliche evolutionäre Algorithmus beginnt.

5.2.3 Fitnessfunktion und Selektion

Um zu bewerten, wie nah ein Bild am Original dran ist, wird der zuvor beschrieben MSE-Algorithmus verwendet. Dafür wird zunächst mithilfe der Bibliothek

pycairo alle Ellipsen auf einen Canvas gezeichnet. Daraufhin wird der Canvas als ein 3-dimensionaler Array gespeichert und die quadrierte Differenz mit dem Array des Originalbildes gebildet. Anschließend wird die Summe berechnet und durch die Anzahl der Pixel - also Breite des Bildes mal die Höhe des Bildes - geteilt. Wichtig dabei ist, dass beim Konvertieren zu einem Array der α-Kanal nicht beachtet wird, das Bild also keine Transparenz mehr besitzt. Die berechnete Fitness wird in der Klassenvariable `fitness` gespeichert. Nachdem die Fitnessfunktion auf alle Individuen angewandt wurde, findet die Selektion statt. In dieser wird die Liste `pictures` aufsteigend nach dem MSE sortiert und dann die ersten 20% für die Rekombination ausgewählt. Aus diesen 20% findet dann die Selektion für die Rekombination statt.

5.2.4 Mutation

Da die Rekombination genau wie im Theorieteil beschrieben implementiert wurde, soll nun gleich die Mutation genauer erläutert werden. Die Mutationswahrscheinlichkeit P_M wurde auf 1 gestellt, sodass jedes Gen mit einer 100%-Wahrscheinlichkeit mutiert wird. Die Stärke der Mutation ist abhängig von den Parametern die mutiert werden sollen. Die RGB-Werte werden um einen zufälligen Wert zwischen -0.075 und 0.075 geändert, die Höhe und y-Position der Ellipsen um $\frac{1}{20}$ der Bildhöhe und die Breite und x-Position um $\frac{1}{20}$ der Bildbreite (beide können sowohl positiv als auch negativ sein). Diese Werte werden außerdem mit 0.995^x multipliziert, wobei x die Nummer der jetzigen Generation ist. Dies wird gemacht, da am Anfang des evolutionären Algorithmus eine höhere und zum Ende hin eine kleine Suchbreite erwünscht ist, um am Anfang das ungefähre Optimum zu finden und am Ende am genauen Optimum anzukommen. Biologisch betrachtet ist also zunächst eine transformierende und zum Schluss eine stabilisierende Selektion von Nöten.

5.2.5 Kompression und Dekompression

Am Ende des evolutionären Algorithmus wird das beste `Picture`-Objekt der letzten Generation ausgewählt, damit seine Ellipsen gespeichert werden können. Eine Speicherung der einzelnen Parameter in Zeichenketten würde sehr viel Platz einnehmen, weswegen sie zunächst binär kodiert werden. Die Farbe der Ellipse, also die RGB-Werte, lassen sich in 3 Bytes speichern. Für die restlichen Parameter werden unsignierte Shorts verwendet. Diese bestehen aus jeweils 2 Bytes, können also $2^{16} = 65536$ verschiedene Ganzzahlen darstellen. Dabei bedeutet „unsigniert“, dass sie kein Vorzeichen speichern, also nur positive Zahlen darstellen können. Diese Werte werden genutzt, um das Verhältnis

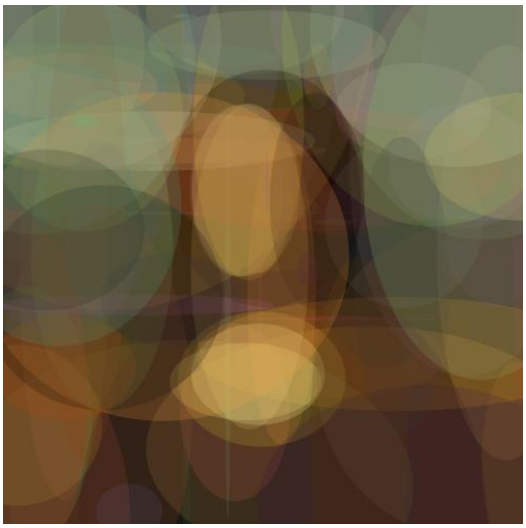
der x-Position und Breite zur Bildbreite und das Verhältnis der y-Position und Höhe zur Bildhöhe darzustellen. Anschließend wir dieser Bruch mit 65536 multipliziert und auf eine Ganzzahl gerundet. Wurden so die Parameter aller Ellipsen in einem Binärstring gespeichert, wird eine neue Datei im selben Verzeichnis wie *main.py* erstellt. In diese Datei wird zunächst die Maße des Bildes in unsignierten Shorts geschrieben und daraufhin der Binärstring mit der Information zu allen Ellipsen. Ganz an den Anfang wird außerdem noch der String „IMAGE “ geschrieben. Dies wird häufig gemacht, damit der Dateityp in der Datei selbst mitenthalten ist. Beispielsweise steht am Anfang jeder PNG-Datei der String „PNG “.
Nachdem die Bilddatei gespeichert wurde, kann die in der Python-Datei *decoder.py* enthaltene Funktion `decode()` aufgerufen werden, um den gesamten Binärstring auszulesen und ihn wieder zurück in eine Liste von Ellipsen zu konvertieren. Danach lassen sich die Ellipsen einfach auf einen Canvas zeichnen und als .png-Datei speichern.

5.3 Auswertung

5.3.1 Durchführung

Jetzt wo die Funktionsweise beschrieben wurde, muss getestet werden, wie gut der Algorithmus in der Praxis die Bewertungskriterien erfüllt. Der Algorithmus wird an einem Datensatz aus 8 bekannten Gemälden und Fotografien getestet. Dabei wird eine Ellipsenanzahl $n = 90$, eine Generationsanzahl $g = 600$ und eine Populationsgröße $p = 100$ verwendet. Alle Originale des Datensatzes und Resultate befinden sich Anhang B. Die Auswertung wird aufgrund von Platzgründen nur anhand der Anwendung auf die Datei "monalisa.png"genauer und visuell erläutert. An den Stellen, wo die Ergebnisse abhängig von der eingelesenen Datei sind, werden die restlichen Werte tabellarisch dargestellt.

(a) Original (monalisa.png)



(b) Ausgabe des Algorithmus

Abbildung 3: Anwendung des Algorithmus auf monalisa.png

5.3.2 Informationsverlust

Wie bereits erläutert, wird für die Fitness-Funktion der Mean-Squared-Error verwendet. Deswegen wird er auch in der Auswertung benutzt, um das Resultat des evolutionären Algorithmus zu beschreiben und später mit einem anderen Algorithmus zu vergleichen. Zunächst soll das Ergebnis für den gesamten Datensatz tabellarisch dargestellt werden. Anhand der hohen Spann-

Datei	MSE	Datei	MSE
monalisa.png	607	ohrring.png	3284
wanderer.png	2606	starrynight.png	2158
goethe.png	2485	lenna.png	2304
tanz.png	2581	afghangirl.png	1970
Durchschnitt:	2249	Spannweite:	2677

Abbildung 4: Ergebnisse des Algorithmus für verschiedene Bilder des Datensatzes

weite lässt sich erkennen, dass die Qualität vom evolutionäre Algorithmus bei unterschiedlichen Bilder sehr stark variiert. Um herauszufinden, an welchen Stellen der Algorithmus Schwierigkeiten hat, wird die Matrix M_{diff} berechnet mit $f(x) = |x|$ und

$$M_{diff} = f \circ (M_{original} - M_{komprimiert})a \tag{4}$$

Dabei entsteht eine neue Matrix, in welcher der RGB-Wert für jeden Pixel durch den Absolutwert der Differenz der RGB-Werte zwischen dem Original

und dem komprimierten Bild entsteht. Stellen wir diese Matrix als Bild dar, erhalten die Abbildung 5a.

(a) Darstellung M_{diff} im RGB-Format (b) Darstellung M_{diff} in Schwarz-Weiß

Abbildung 5: Differenzmatrix für monalisa.png in Bild-Form

In diesem Bild stellt die Helligkeit ein Maß dafür da, wie nah das Ergebnis am Original ist. Das liegt daran, dass die RGB-Werte bei einer größeren Differenz höher sind und damit heller. Um die Helligkeit genauer zu erkennen, kann das Bild wie in Abbildung 5b zu Schwarz-Weiß konvertiert werden, indem die Funktion $f(R, G, B) = (\frac{R+G+B}{3}, \frac{R+G+B}{3}, \frac{R+G+B}{3})$ auf alle Pixel angewendet wird. Dies produziert ein Schwarz-Weiß-Bild, in welchem dunkle Flächen für höhere und helle Flächen für niedrigere Genauigkeit stehen. Wie in der Hypothese vermutet, stechen in Abbildung 5b flächenmäßig kleine Details wie die Augen oder ein Teil der Nase hervor. Auch feinere Strukturen, wie die in ihrem Haar und im See (links mittig im Bild), zeigen stärkere Abweichungen. Damit lässt sich die hohe Spannweite erklären. Bilder mit vielen Details, die farbig hervorstechen, oder Bilder mit hohem Wert auf Struktur, wie die Sternennacht von van Gogh können weniger erfolgreich in Ellipsen dargestellt werden als Bilder, die viele ähnlich gefärbte Flächen aufweisen, wie die Mona Lisa - vor allem lässt sich das im Hintergrund bei den Feldern, Wäldern und Himmel erkennen. Weitergehend lässt sich auch die Optimierung abhängig von der Zahl der vergangenen Generationen auswerten. Dafür kann ein MSE-Generationen-Diagramm erstellt werden (Abbildung 6). Der Verlauf der Kurve beschreibt annähernd eine verschobene Exponentialfunktion mit einer Basis, deren Betrag kleiner als 1 ist. Die Änderung des MSE abhängig von der Generation ist die Ableitung der Funktion und demnach auch eine Exponentialfunktion. Das bedeutet für den Algorithmus, dass die größten Veränderung am Anfang statt-

finden und er nach einer Zeit stagniert, auch wenn das Optimum noch nicht ganz erreicht wurde. Daraus lässt sich schlussfolgern, dass das Ergebnis sich nach Generation 600 nur sehr geringfügig ändern würde. Ein visueller Einblick in den Verlauf der Evolution befindet sich im Anhang B.

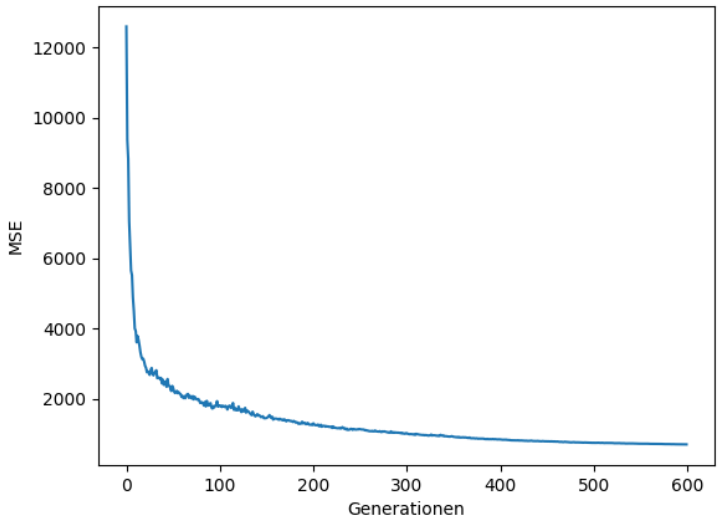


Abbildung 6: MSE-Generationen-Diagramm für monalisa.png

5.3.3 Kompression

Der eingenommene Speicherplatz lässt sich, anders als die Genauigkeit zum Original, genau berechnen. Am Anfang der Dateien werden 2 Mal 2 Bytes für die Höhe und Breite des Bildes verwendet. Dann werden für jede Ellipse 4 Bytes für die Position, 4 Bytes für die Halbachsen und 3 Bytes für die Farbe gespeichert. Optional kann am Anfang der Datei auch ein String der Länge l geschrieben werden, um das Dateiformat zu erkennen, welcher 1 Byte pro Zeichen einnimmt. Damit ergibt sich die Speicherkomplexität (in Bytes) bei n Ellipsen als

$$S(n) = l + 4 + n \cdot 11 \tag{5}$$

Der Speicheraufwand ist also direkt proportional zu der Ellipsenanzahl n und nicht von p oder g abhängig. Setzt man alle für den Versuch genutzen Werte ein, kommt man auf $5 + 4 + 90 \cdot 11 = 999$ Bytes ≈ 1 Kilobyte, was eine noch größere Kompression als erwartet darstellt. Vergleicht man nun den Speicherplatz der Originalbildes mit dem der generierten Bilder, kommt man auf in Abbildung 7 dokumentierten Kompressionsraten (welche das Verhältnis $\frac{S(Original)}{S(Komprimiert)}$ beschreiben), die alle im mittleren dreistelligen Bereich liegen.

Datei	Kompressionsrate	Datei	Kompressionsrate
monalisa.png	380	ohrring.png	331
wanderer.png	302	starrynight.png	685
goethe.png	603	lenna.png	463
tanz.png	566	afghangirl.png	606

Abbildung 7: Kompressionsraten der Resultate

5.3.4 Laufzeit

Die Laufzeitkomplexität soll zunächst theoretisch betrachtet werden. Während eines Durchlaufs wird über die Anzahl der Generationen g iteriert. In dieser Schleife wird zunächst eine Fitnessfunktion auf alle p Individuen angewandt, in der jede der n Ellipsen auf einen Canvas gezeichnet wird (der Einfachheit halber soll angenommen werden, dass das Zeichnen einer Ellipse $O(1)$, also von konstanter Laufzeitkomplexität, ist und nicht abhängig von ihrer Größe oder Position). Nach dem Aufrufen der Fitnessfunktionen werden die Individuen nach ihrer Fitness sortiert, was eine Laufzeitkomplexität von $p \log p$ besitzt (da der Sortieralgorithmus Timsort verwendet wird [15]). Bei der darauf folgenden Rekombination wird p Mal ein neues Individuum erstellt. Nach dem Ende des evolutionären Algorithmus wird ein letztes Mal über alle Ellipsen des besten Individuums iteriert, um sie binär zu speichern. In der O-Notation gilt für $O(n, p, g)$ somit:

$$O(n, p, g) = g \cdot (p \cdot n + p \log p + p) + n \tag{6}$$

Klammert man p aus, erhält man:

$$O(n, p, g) = g \cdot (p \cdot (n + \log p + 1)) + n \tag{7}$$

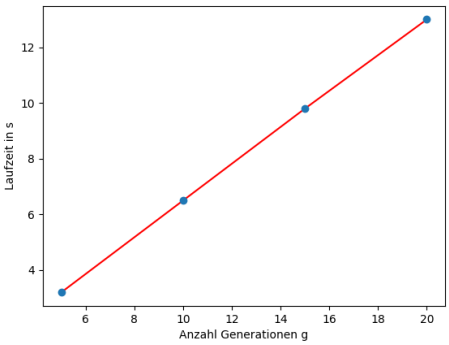
$$O(n, p, g) = g \cdot (p \cdot (n + \log p)) + n \tag{8}$$

Da konstante Werte in der O-Notation vernachlässigt werden können, wird die 1 im Term ausgelassen. Zudem kann auch das $+n$ am Ende des Terms im Vergleich zu $\cdot(n + \log p)$ für $n, p \in \mathbb{N}^+$ vernachlässigt werden kann. Somit kommt man auf folgende Laufzeitkomplexität:

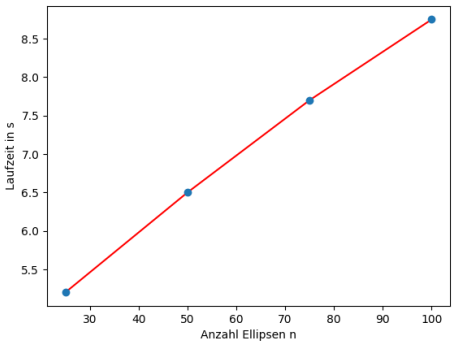
$$O(n, p, g) = g \cdot p \cdot (n + \log p) \tag{9}$$

Hierbei sei noch anzumerken, dass für $n \approx g$ sich der Term $(n + \log p)$ an n annähert, da die Logarithmusfunktion sehr langsam steigt. Es ist also zu erwarten, dass der $\log p$ Ausdruck auf die folgenden Messungen keinen starken

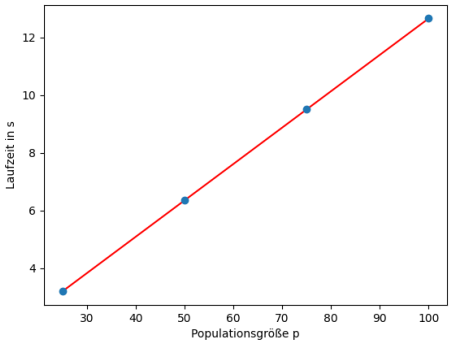
Einfluss haben wird.
Aus dieser theoretischen Laufzeitanalyse lässt sich erschließen, dass mit linear steigendem n, p und g auch die Laufzeit in etwa linear steigen wird. Um diese theoretische Betrachtung zu überprüfen, wird die Laufzeit in Abhängigkeit einer der 3 Variablen gemessen, wobei die beiden anderen Variablen konstant gehalten werden. Für das Untersuchen jeder Variable wurden jeweils 4 Messpunkte genutzt.



(a) Abhängigkeit von g
$(n = 50, p = 50)$



(b) Abhängigkeit von n
$(p = 50, g = 10)$



(c) Abhängigkeit von p
$(n = 50, g = 10)$

Abbildung 8: Abhängigkeit der Laufzeit von n, g, p

In den Diagrammen 8a und 8c kann man sehr gut erkennen, dass eine Gerade durch die 4 Punkte geht, die Vermutung also zutrifft. Beim Diagramm 8b scheint es sich jedoch einer Logarithmusfunktion anzunähern. Erweitert man aber das Diagramm um drei weitere Messpunkte (Diagramm 9), so wird klar, dass die Laufzeit sich nur sehr irregulär verhält. Dies könnte daran liegen, dass *pycairo* die Ellipsen nicht konstant schnell zeichnet. Was sich auf jeden Fall feststellen lässt, ist, dass die Funktion monoton steigend ist, also mit einer höheren Ellipsenanzahl die Laufzeit immer steigt, auch wenn kein eindeutiges Verhältnis festgestellt werden kann.

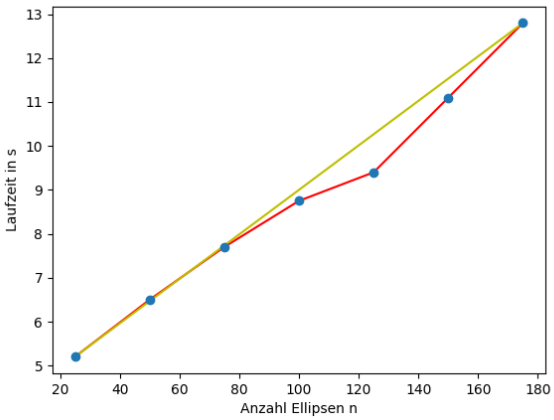


Abbildung 9: Erweitertes Laufzeit-n-Diagramm

5.3.5 Vergleich mit JPEG

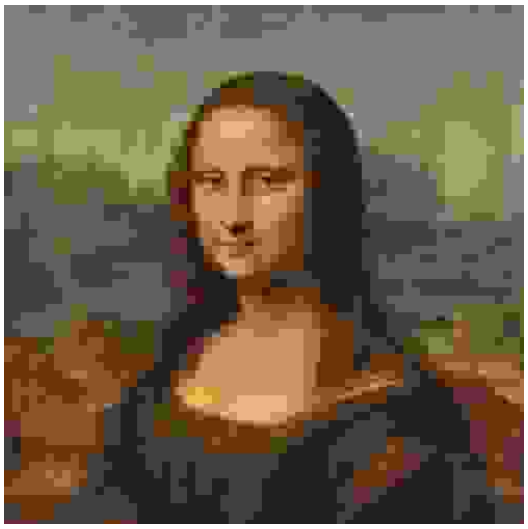
Jetzt wo der Algorithmus in mehreren Aspekten ausgewertet wurde, soll er mit dem häufig eingesetzten Kompressionsalgorithmus JPEG verglichen werden. Dieser besitzt mehrere Kompressionsstufen zwischen 1 und 100, wobei 1 die größte Kompression und kleinste Genauigkeit bedeutet. Der MSE und Speicherplatz der Resultate soll mit dem selbst entwickelten Algorithmus verglichen werden. Dabei wird vor allem Wert auf die größte Kompressionsstufe 1 gelegt, um zu schauen, was die maximale mögliche Kompressionsrate ist. Dabei stellt sich raus, dass der Speicherplatz der von JPEG komprimierten Bilder zwischen 2 und 4 Kilobyte liegt, also immernoch um Vielfache größer als die Ergebnisse des evolutionären Algorithmus. Wird nun der MSE verglichen und tabellarisch dargestellt - wobei ein positiver Wert bedeutet, dass der MSE des JPEG-komprimierten Bildes größer war und umgekehrt - erhält man die Tabelle in Abbildung 10. An den Ergebnisse kann man erkennen, dass die

Datei	MSE	Datei	MSE
monalisa.png	+488	ohrring.png	-2351
wanderer.png	-1799	starrynight.png	+635
goethe.png	-1048	lenna.png	-866
tanz.png	-1257	afghangirl.png	-711

Abbildung 10: MSE der JPEG-Kompression als Vergleich zur Kompression des eigenen Algorithmus

JPEG-Kompression in den meisten Fällen ein viel besseres Ergebnis produziert als die Kompression des eigenen Algorithmus. Zudem kann man in den von JPEG erzeugten Bildern oft mehr Details (wie den Gesichtsausdruck) erkennen - sogar in den Fällen, wo der MSE größer ist, also die Farben teils

ungenauer als beim evolutionären Algorithmus (Abbildung 11a). Zudem kann JPEG-Kompression auch bei nur 18 Kilobyte Speicherplatz ein vom Original kaum zu unterscheidbares Ergebnis produzieren (Abbildung 11b).



(a) JPEG-Kompression Level 1

(b) JPEG-Kompression Level 50

Abbildung 11: Ausgaben des JPEG-Algorithmus bei Level 1 und 50

6 Fazit

6.1 Verbesserungsansätze

Abschließend lässt sich sagen, dass in dieser Arbeit durch das Implementieren eines eigenen Programms und Testen dieses gezeigt werden konnte, dass evolutionäre Algorithmen durchaus in diesem Bereich eingesetzt werden können und sehr hohe Kompressionsraten ermöglichen können. Die beiden größten Schwachstellen sind allerdings, dass Strukturen und kleine Objekte in den Bildern nicht erfasst werden und die sehr hohe Laufzeit. In zukünftigen Arbeiten könnten verschiedene geometrische Formen getestet werden und ihre Genauigkeit verglichen werden. Außerdem könnten andere Algorithmen für die Fitnessfunktion verwendet werden. Hier wurde der MSE-Algorithmus genutzt, man könnte aber auch den sogenannten SSM (Structural Similarity Measure)-Algorithmus anwenden, um zu schauen, ob sich die Ergebnisse verbessern. Für die Verbesserung der Laufzeit könnte man das Programm in schnelleren Programmiersprachen, wie C++, implementieren, da Python zu den langsameren gehört.
Diese Ideen könnten die jetzigen Schwachstellen minimieren. Ein letzter Ansatz, der die Kompressionsrate sogar noch weiter vergrößern kann, wäre, für

die Parameter der Ellipsen - also Position und Länge der Halbachsen - nicht 2 Bytes sondern jeweils 1.5 Bytes zu nutzen. Damit würde selbst bei Bildern mit einer Auflösung von 4096x4096 (2^{12} x 2^{12}) keine Verluste bei der Speicherung der Ellipsen entstehen und bei höheren Ellipsen wären die Abweichungen nur schwer merkbar. Der Vorteil wäre, dass 2 Bytes pro Ellipse gespart werden würden. Der Grund, warum dieser Ansatz nicht implementiert wurde, ist, dass eine manuelle Speicherung von Bits anstatt von Bytes sehr viel komplexer ist und vom zeitlichen Aufwand nicht tragbar.

6.2 Ausblick

Abgesehen davon, ob die genannten Verbesserungsansätze sinnvoll umgesetzt werden können und zu Verbesserungen führen würde, steht fest, dass der Speicherplatz der komprimierten Bilder extrem gering ist, was man auch beim Vergleich mit dem Kompressionsverfahren JPEG gemerkt hat. Die Frage besteht nun leider darin, ob eine hohe Kompressionsrate wirklich einen genauso hohen Informationsverlust rechtfertigt, vor allem wenn JPEG bei einer ähnlich bemerkenswerten Kompression das Original fast 1:1 wiedergibt. Dies ist auch zu erwarten, wenn man bedenkt, dass JPEG über Jahre optimiert wurde und bei der Seminararbeit viele zeitliche und technische Limitationen vorliegen. Nichtsdestotrotz kann das Programm beispielsweise genutzt werden, wenn sehr viele Daten gleichzeitig versendet werden müssen, da dabei selbst eine doppelt so starke Kompression einen Einfluss haben kann. So können Ergebnisse des evolutionären Algorithmus als Vorschau-Bilder in der Google-Bildersuche Anwendung finden, weil so selbst bei langsamen Internet mehrere Bilder gleichzeitig geladen werden können. Erst beim Auswählen eines der Vorschau-Bilder würde dann das Original geladen werden. In diesem Beispiel würde sogar die hohe Laufzeit keine großen Einschränkungen hervorrufen, da die Bilder nur ein einziges Mal komprimiert werden müssten und die Dekompression wenige Millisekunden benötigt.
Ob evolutionäre Algorithmen in Zukunft wirklich für solche Zwecke verwendet werden, steht bisher in den Sternen, aber vielleicht müssen sie, wie Kunst generell, keine spezielle Anwendung haben, sondern einfach faszinierende Abbildungen der Realität erschaffen und unser Verständnis der Welt vorantreiben.

7 Literatur und Abbildungsverzeichnis

Literaturverzeichnis

[1] M. Mitchell, *An introduction to genetic algorithms*, 7. print, Ser. Complex adaptive systems. Cambridge, Mass., 2001, 209 S., OCLC: 256769418, ISBN: 978-0-262-63185-3 978-0-262-13316-6.

[2] J. Sung und B. Jeong, "An adaptive evolutionary algorithm for traveling salesman problem with precedence constraints," *The Scientific World Journal*, Jg. 2014, e313767, 17. Feb. 2014, Publisher: Hindawi, ISSN: 2356-6140. DOI: 10.1155/2014/313767. Adresse: https://www.hindawi.com/journals/tswj/2014/313767/ (besucht am 08. 04. 2021).

[3] F. Pezzella, G. Morganti und G. Ciaschetti, "A genetic algorithm for the flexible job-shop scheduling problem," *Computers & Operations Research*, Part Special Issue: Search-based Software Engineering, Jg. 35, Nr. 10, S. 3202–3212, 1. Okt. 2008, ISSN: 0305-0548. DOI: 10.1016/j.cor.2007.02.014. Adresse: https://www.sciencedirect.com/science/article/pii/S0305054807000524 (besucht am 08. 04. 2021).

[4] A. Ghaheri, S. Shoar, M. Naderan und S. S. Hoseini, "The Applications of Genetic Algorithms in Medicine," *Oman Medical Journal*, Jg. 30, Nr. 6, S. 406–416, Nov. 2015, ISSN: 1999-768X. DOI: 10.5001/omj.2015.82. Adresse: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4678452/ (besucht am 08. 04. 2021).

[5] S. Mirjalili, J. Song Dong, A. S. Sadiq und H. Faris, "Genetic algorithm: Theory, literature review, and application in image reconstruction," in *Nature-Inspired Optimizers: Theories, Literature Reviews and Applications*, Ser. Studies in Computational Intelligence, S. Mirjalili, J. Song Dong und A. Lewis, Hrsg., Cham: Springer International Publishing, 2020, S. 69–85, ISBN: 978-3-030-12127-3. DOI: 10.1007/978-3-030-12127-3_5. Adresse: https://doi.org/10.1007/978-3-030-12127-3_5 (besucht am 09. 04. 2021).

[6] A. Bilsing, K.-H. Firtzlaff, K.-H. Gehlhaar, E. Kemnitz, M. Kurze, L. Naunpapper, C. Pews-Hocke, H. Simon und E. Zabel, *Biologie: 5. bis 10. Klasse*, 3., aktualisierte Auflage, Ser. Duden, Schulwissen ; Biologie. Berlin: Dudenverlag, 2015, 400 S., ISBN: 978-3-411-73593-8.

[7] (4. Juli 2018). "19: The evolution of populations," Biology LibreTexts, Adresse: https://bio.libretexts.org/Bookshelves/Introductory_and_General_Biology/Book%3A_General_Biology_(Boundless)/19%3A_The_Evolution_of_Populations (besucht am 26. 08. 2021).

[8] B. Hill, *Bionik. Lehrbuch*, 1. Aufl., [Nachdr.], Ser. Natur, Mensch, Technik Bionik, Lehrbuch. Berlin Mannheim: Duden Paetec Schulbuchverl, 2009, 96 S., ISBN: 978-3-8355-3018-8.

[9] R. Poli, W. Langdon und N. Mcphee, *A Field Guide to Genetic Programming*. 1. Jan. 2008, ISBN: 978-1-4092-0073-4.

[10] J. Sachs, "Digital Image Basics," *Digital Light & Color*, 1996. Adresse: http://www.microscopist.co.uk/wp-content/uploads/2017/04/digital-basics.pdf (besucht am 05. 10. 2021).

[11] T. Porter, "Computer graphics volume 18," S. 7, 1984.

[12] Zhou Wang und A. Bovik, "Mean squared error: Love it or leave it? a new look at signal fidelity measures," *IEEE Signal Processing Magazine*, Jg. 26, Nr. 1, S. 98–117, Jan. 2009, ISSN: 1053-5888. DOI: 10.1109/MSP.2008.930649. Adresse: http://ieeexplore.ieee.org/document/4775883/ (besucht am 05. 10. 2021).

[13] . ScholarSpace -. (11. Feb. 2021). "Research guides: All about images: Raster vs. vector images," Adresse: https://guides.lib.umich.edu/c.php?g=282942&p=1885352 (besucht am 06. 10. 2021).

[14] R. Johansson. (7. Dez. 2008). "Genetic programming: Evolution of mona lisa," Roger Johansson Blog, Adresse: https://rogerjohansson.blog/2008/12/07/genetic-programming-evolution-of-mona-lisa/ (besucht am 07. 10. 2021).

[15] N. Auger, V. Jugé, C. Nicaud und C. Pivoteau, "On the Worst-Case Complexity of TimSort," *arXiv:1805.08612 [cs]*, 7. Juli 2019. arXiv: 1805.08612. Adresse: http://arxiv.org/abs/1805.08612 (besucht am 02. 10. 2021).

Abbildungsverzeichnis

1 grafische Darstellung der 3 Selektionstypen 2
2 RGB-Farbraum als dreidimensionales Koordinatensystem 7
3 Anwendung des Algorithmus auf monalisa.png 14
4 Ergebnisse des Algorithmus für verschiedene Bilder des Datensatzes 14
5 Differenzmatrix für monalisa.png in Bild-Form 15
6 MSE-Generationen-Diagramm für monalisa.png 16
7 Kompressionsraten der Resultate 17
8 Abhängigkeit der Laufzeit von n, g, p 18
9 Erweitertes Laufzeit-n-Diagramm 19
10 MSE der JPEG-Kompression als Vergleich zur Kompression des eigenen Algorithmus 19
11 Ausgaben des JPEG-Algorithmus bei Level 1 und 50 20
12 monalisa.png (Mona Lisa) 32
13 ohrring.png (Das Mädchen mit dem Perlenohrgehänge) 32
14 wanderer.png (Der Wanderer über dem Nebelmeer) 33
15 starrynight.png (Sternennacht) 33
16 goethe.png (Goethe in der Campagna) 33
17 lenna.png (Lenna - Fotografie, die häufig als Testbild für Bildverarbeitung genutzt wird 34
18 tanz.png (Der Tanz) 34
19 afghangirl.png (Afghanisches Mädchen) 34
20 Evolution der Mona Lisa 35

Alle Abbildungen wurden eigenständig erstellt. Die Bilder für den Datensatz stammen von Wikimedia.

A Quellcode

```
# Import aller wichtigen Bibliotheken
from random import random as rnd
import cairo
import math
import numpy as np
from PIL import Image
from struct import pack

# Bild wird als Numpy-Array geladen
file = "monalisa.png"
image = Image.open(file)
image = image.convert('RGB')
data = np.asarray(image).astype("float")

# Funktion um eine Variable zwischen einem Minimum und Maximum zu beschränken
def constr(n,minim,maxim):
    return max(minim,min(n,maxim))

# Klasse Ellipse, die alle Parameter einer Ellipse beinhaltet
class Ellipse():
    def __init__(self,screen):
        self.color = [0,0,0,0]
        self.x = 0
        self.y = 0
        self.r1 = 0
        self.r2 = 0
        self.screen = screen

    # Mutationsfunktion, die alle Werte zufällig ändert
    def mutate(self,rate):
        # damit die Ellipsen im Bildschirm bleiben und die RGB-Werte
        # zwischen 0 und 1 liegen, wird die constr-Funktion benutzt
        # Anmerkung: pycairo liest die RGB-Werte zwischen 0 und 1 ein,
        # es gibt aber immernoch nur 256 mögliche Werte für R,G und B

        # r,g,b
        self.color[0]=constr(self.color[0]+rate*0.15*(rnd()-0.5),0,1)
        self.color[1]=constr(self.color[1]+rate*0.15*(rnd()-0.5),0,1)
```

```
39          self.color[2]=constr(self.color[2]+rate*0.15*(rnd()
    -0.5),0,1)
40          # x,y
41          self.x=constr(self.x+(rnd()-0.5)*rate*self.screen
    [0]/20,1,self.screen[0])
42          self.y=constr(self.y+(rnd()-0.5)*rate*self.screen
    [1]/20,1,self.screen[1])
43          # r1,r2
44          self.r1=constr(self.r1+(rnd()-0.5)*rate*self.screen
    [0]/20,1,self.screen[0])
45          self.r2=constr(self.r2+(rnd()-0.5)*rate*self.screen
    [1]/20,1,self.screen[1])
46
47      # Generation einer zufälligen Ellipse
48      def random(self):
49          self.color = [rnd(),rnd(),rnd(),0.2]
50          self.x,self.y = rnd()*self.screen[0],rnd()*self.screen
    [1]
51          self.r1,self.r2 = rnd()*self.screen[0]/3,rnd()*self.
    screen[1]/3
52
53  # Klasse Picture in der sich eine bestimmte Anzahl von Ellipsen
     befindet
54  class Picture():
55      def __init__(self,screen_size,ellipses):
56          self.screen_size = screen_size
57          self.fitness = 0
58          self.ellipses = ellipses
59
60      # randomisiert alle Ellipsen des Bildes
61      def random(self):
62          for ellipse in self.ellipses:
63              ellipse.random()
64
65      # mutiert alle Ellipsen des Bildes
66      def mutate(self,rate):
67          for ellipse in self.ellipses:
68              ellipse.mutate(rate)
69
70      # Fitness-Funktion
71      def fitness_func(self):
72          # erstellt den pycairo-Canvas
73          width,height = self.screen_size
74          ims = cairo.ImageSurface(cairo.FORMAT_ARGB32, width,
    height)
75          cr = cairo.Context(ims)
76
```

```
77          # zeichnet alle Ellipsen auf den Canvas
78          for ellipse in self.ellipses:
79              cr.save()
80              r,g,b = ellipse.color[0],ellipse.color[1],ellipse.
    color[2]
81              a = ellipse.color[3]
82              cr.set_source_rgba(r, g, b, a)
83              x,y,r1,r2 = ellipse.x,ellipse.y,ellipse.r1,ellipse.
    r2
84              cr.translate(x,y)
85              cr.scale(r1,r2)
86              cr.arc(0, 0, 1, 0, 2*math.pi)
87              cr.fill()
88              cr.restore()
89
90          # berechnet die Fitness des Bildes mithilfe des MSE
91          buf = ims.get_data()
92          nparr = np.ndarray(
93                  shape=(self.screen_size[0],self.screen_size
    [1],4),
94                  dtype=np.uint8,buffer=buf)[:,:,:3]
95          err = np.sum((nparr.astype("float") - data) ** 2)
96          err /= float(nparr.shape[0] * nparr.shape[1])
97          self.fitness = err
98
99      # speichert das derzeitige Bild
100     def output(self,gen_num):
101         width,height = self.screen_size
102         ims = cairo.ImageSurface(cairo.FORMAT_ARGB32, width,
    height)
103         cr = cairo.Context(ims)
104
105         for ellipse in self.ellipses:
106             cr.save()
107             r,g,b = ellipse.color[0],ellipse.color[1],ellipse.
    color[2]
108             a = ellipse.color[3]
109             cr.set_source_rgba(r, g, b, a)
110             x,y,r1,r2 = ellipse.x,ellipse.y,ellipse.r1,ellipse.
    r2
111             cr.translate(x,y)
112             cr.scale(r1,r2)
113             cr.arc(0, 0, 1, 0, 2*math.pi)
114             cr.fill()
115             cr.restore()
116
117         buf = ims.get_data()
```

```
118         nparr = np.ndarray(
119             shape=(self.screen_size[0],self.screen_size[1],4),
120             dtype=np.uint8, buffer=buf)[:,:,:3]
121         Image.fromarray(nparr).save("output"+str(gen_num)+".jpg
    ")
122
123 # Klasse für die Population
124 class Population():
125     def __init__(self,pictures):
126         self.pictures = pictures
127         self.screen_size = self.pictures[0].screen_size[:]
128         self.best = None
129         for picture in pictures:
130             picture.random()
131
132     # Simulierung der Generationen
133     def run(self,n):
134         for gen in range(n):
135             self.generation(gen)
136             self.best.output(gen)
137
138     # Selektion, Rekombination, Mutation und Erstellung einer
139     # neuen Population während einer Generation
140     def generation(self,gen):
141         gene_pool = []
142         #Selektion
143         self.best = [0,None]
144
145         for id,picture in enumerate(self.pictures):
146             picture.fitness_func()
147             # Speicherung des Besten aus der Population
148             if picture.fitness>self.best[0]:
149                 self.best = [picture.fitness,picture]
150             gene_pool.append((id, picture.fitness))
151
152         gene_pool = sorted(gene_pool, key=lambda x: x[1])
153         self.best = self.best[1]
154         new_pictures = []
155
156         # Rekombination
157         for _ in range(len(self.pictures)-1+1):
158             new_picture = Picture(self.screen_size[:],[])
159             # Top 20% der Population weren für die Eltern
    selektiert
160             P1,P2=(self.pictures[gene_pool[int(rnd()*len(
    gene_pool)*0.2)][0]],
161                    self.pictures[gene_pool[int(rnd()*len(
```

```
    gene_pool)*0.2)][0]])
162             for id in range(len(P1.ellipses)):
163                 new_ellipse = Ellipse(self.screen_size[:])
164                 # jede Ellipse wird zufällig von einem der
    Elternpaare gewählt
165                 p1 = rnd()>0.5
166                 if p1: new_ellipse.x = P1.ellipses[id].x
167                 else:  new_ellipse.x = P2.ellipses[id].x
168 
169                 if p1: new_ellipse.y = P1.ellipses[id].y
170                 else:  new_ellipse.y = P2.ellipses[id].y
171 
172                 if p1: new_ellipse.r1 = P1.ellipses[id].r1
173                 else:  new_ellipse.r1 = P2.ellipses[id].r1
174 
175                 if p1: new_ellipse.r2 = P1.ellipses[id].r2
176                 else:  new_ellipse.r2 = P2.ellipses[id].r2
177 
178                 if p1: new_ellipse.color[0] = P1.ellipses[id].
    color[0]
179                 else:  new_ellipse.color[0] = P2.ellipses[id].
    color[0]
180 
181                 if p1: new_ellipse.color[1] = P1.ellipses[id].
    color[1]
182                 else:  new_ellipse.color[1] = P2.ellipses[id].
    color[1]
183 
184                 if p1: new_ellipse.color[2] = P1.ellipses[id].
    color[2]
185                 else:  new_ellipse.color[2] = P2.ellipses[id].
    color[2]
186 
187                 if p1: new_ellipse.color[3] = P1.ellipses[id].
    color[3]
188                 else:  new_ellipse.color[3] = P2.ellipses[id].
    color[3]
189                 new_picture.ellipses.append(new_ellipse)
190             # Mutation
191             new_picture.mutate(0.995**gen)
192             new_pictures.append(new_picture)
193         self.pictures = new_pictures[:]
194 
195 # ------------------------------------------------------
196 
197 # Ausführung der Generation mit 100 Individuen, 90 Ellipsen
198 # und 600 Generationen
```

```
199 p,n,g = 100,90,600
200 p = Population([Picture(image.size[:],[Ellipse(image.size[:])
201 for _ in range(n)])for _ in range(p)])
202 p.run(g)
203
204 # Auswahl des Besten Individuums
205 result = p.best
206
207 # Erstellung der Binärdatei (.eai = evolutionärer
      Kompressionsalgorithmus)
208 output = open("output.eka","wb")
209 binary1,binary2 = bytearray(),bytearray()
210 output.write(bytearray("IMAGE",encoding="ascii"))
211 output.write(pack("HH",result.ellipses[0].screen[0],result.
      ellipses[0].screen[1]))
212
213 # Speicherung der Ellipsen in Binärform
214 for ellipse in result.ellipses:
215     # Speicherung der Koordinaten und Radien in Shorts
216     binary1+=pack("HHHH",
217         int(ellipse.x/ellipse.screen[0]*65535),
218         int(ellipse.y/ellipse.screen[1]*65535),
219         int(ellipse.r1/ellipse.screen[0]*65535),
220         int(ellipse.r2/ellipse.screen[1]*65535)
221     )
222     # Speicherung der RGB-Werte in Bytes
223     binary2+=pack("BBB",
224         int(ellipse.color[0]*255),
225         int(ellipse.color[1]*255),
226         int(ellipse.color[2]*255)
227     )
228
229 # Ausgabe der komprimierten Binär-Datei
230 output.write(binary1+binary2)
```

Datei 1: main.py

```
1 # Import aller wichtigen Bibliotheken
2 from struct import unpack
3 import numpy as np
4 import math
5 from PIL import Image
6 import cairo
7
8 def decode(file):
9     # Einlesen der Binärdatei
10     binary = file.read()
11     # Überprüfen, ob die ersten 5 Bytes stimmen
```

```
12      if binary[:5].decode(encoding="ascii")!="IMAGE":
13          print("Es wurde ein falsches Dateiformat ausgewählt")
14          return
15      binary = binary[5:]
16      num_ellipses = (len(bytearray(binary))-4)//11
17
18      # Binärstring in einen Array aus Integern konvertieren
19      data = unpack("HH"+"HHHH"*num_ellipses+"BBB"*num_ellipses,
    binary)
20      width,height = data[0],data[1]
21      # Cairo-Canvas erstellen
22      ims = cairo.ImageSurface(cairo.FORMAT_ARGB32, width, height
    )
23      cr = cairo.Context(ims)
24      data = data[2:]
25
26      # alle Ellipsen aus den Informationen der eingelesenen
    Datei generieren
27      for i in range(num_ellipses):
28              x,y,r1,r2 = data[i*4:i*4+4]
29              x,y = x/65535*width,y/65535*height
30              r1,r2 = r1/65535*width,r2/65535*height
31              r,g,b = data[num_ellipses*4+i*3:num_ellipses*4+i
    *3+3]
32              r,g,b = r/255,g/255,b/255
33              cr.save()
34              cr.set_source_rgba(r, g, b, 0.2)
35              cr.translate(x,y)
36              cr.scale(r1,r2)
37              cr.arc(0, 0, 1, 0, 2*math.pi)
38              cr.fill()
39              cr.restore()
40
41      # Speichern der Datei als JPG-Datei
42      buf = ims.get_data()
43      nparr = np.ndarray (shape=(width,height,4), dtype=np.uint8,
        buffer=buf)[:,:,:3]
44      Image.fromarray(nparr).save("decompressed.png")
45
46  #Dekodieren der ausgewählten Datei
47  file = "output.eka"
48  decode(open(file,"rb"))
```

Datei 2: decoder.py

B Ergebnisse

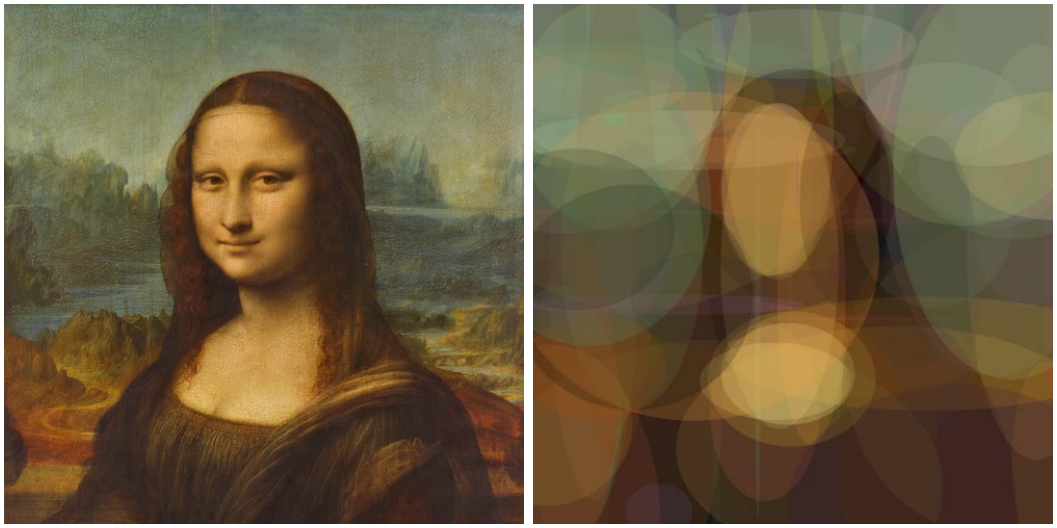


Abbildung 12: monalisa.png (Mona Lisa)

Abbildung 13: ohrring.png (Das Mädchen mit dem Perlenohrgehänge)

Abbildung 14: wanderer.png (Der Wanderer über dem Nebelmeer)

Abbildung 15: starrynight.png (Sternennacht)

Abbildung 16: goethe.png (Goethe in der Campagna)

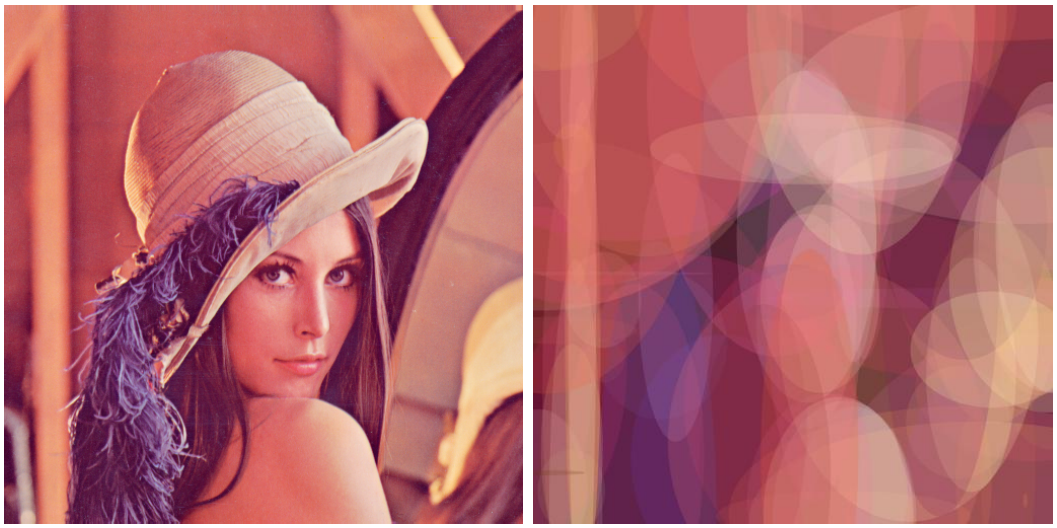


Abbildung 17: lenna.png (Lenna - Fotografie, die häufig als Testbild für Bildverarbeitung genutzt wird

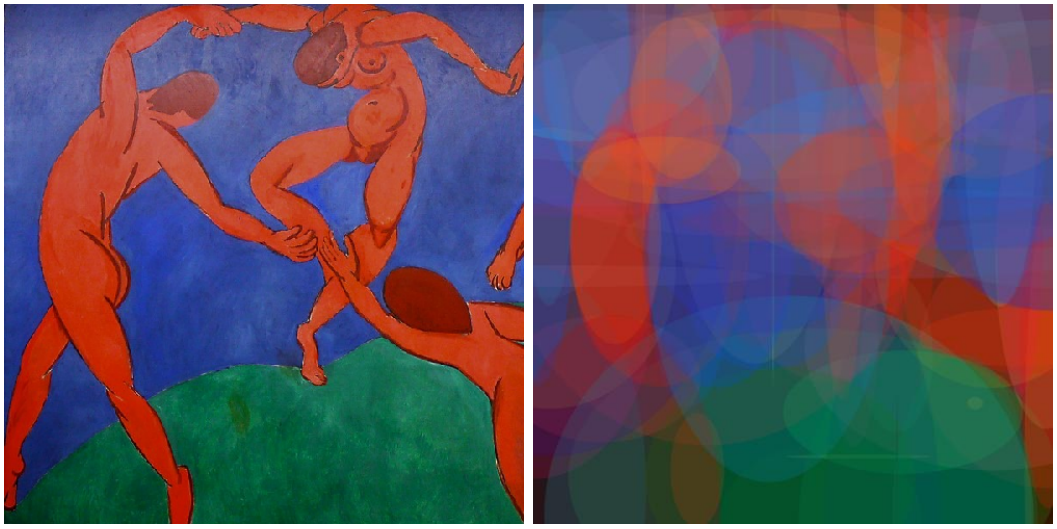
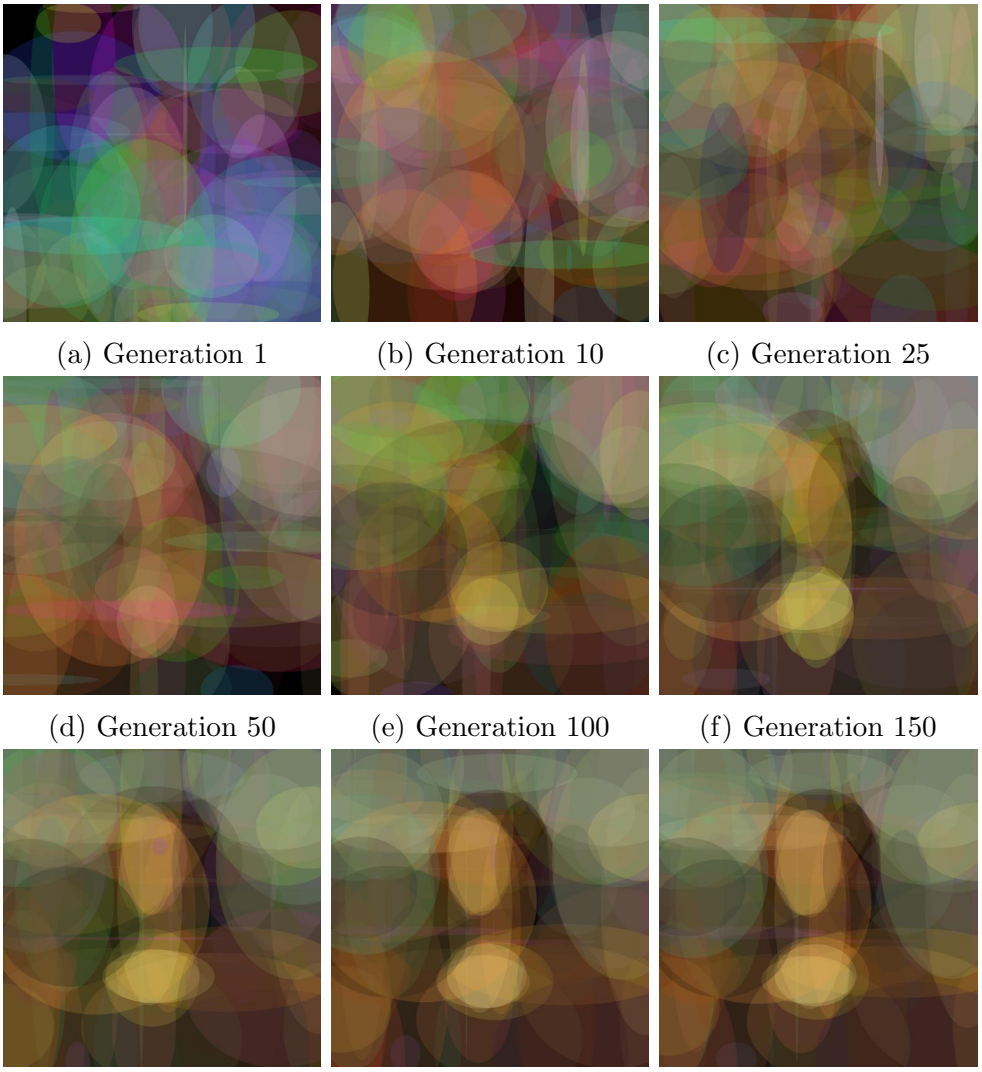


Abbildung 18: tanz.png (Der Tanz)

Abbildung 19: afghangirl.png (Afghanisches Mädchen)

9 Ausschnitte aus der Evolution des „Mona Lisa“-Bildes:



(a) Generation 1 (b) Generation 10 (c) Generation 25

(d) Generation 50 (e) Generation 100 (f) Generation 150

(g) Generation 300 (h) Generation 450 (i) Generation 600

Abbildung 20: Evolution der Mona Lisa

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Zuhilfenahme anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe. Die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen sind als solche kenntlich gemacht.

Unterschrift: _____

Potsdam, 08.10.2021