

[JUNGE] wissenschaft

Jungforscher*Innen publizieren
online | **peer reviewed** | original

Verlag:
Physikalisch-
Technische
Bundesanstalt



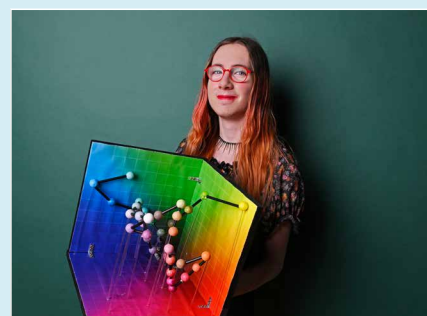
Mathematik &
Informatik

Die perfekte Reihenfolge

Sortierung mehrdimensionaler Objekte als
graphentheoretisches und algorithmisches Problem

Bücher nach der Farbe des Buchrückens zu sortieren und den kürzesten Weg zu finden, der verschiedene Orte verbindet, haben eines gemeinsam: mehrdimensionale Sortierung. Die Arbeit zeigt, dass die eindimensionale Sortierung ein Spezialfall n -dimensionaler Sortierung ist, äquivalent zum NP-schweren Problem des kürzesten Hamilton-Pfads. Eine interaktive Webanwendung veranschaulicht den Beweis und erlaubt es, verschiedene mehrdimensionale Objekte mit unterschiedlichen Algorithmen selbst zu sortieren.

DIE JUNGFORSCHERIN



© Stiftung Jugend forscht e.V.

Leo Blume (*2008)
Gymnasium Essen-Werden

Eingang der Arbeit:
10.9.2024

Arbeit angenommen:
2.2.2025



Die perfekte Reihenfolge

Sortierung mehrdimensionaler Objekte als graphentheoretisches und algorithmisches Problem

1. Einleitung, Fragestellung und Vorgehensweise

Wie kann man Bücher nach Farben sortieren? Diese Frage stellte ich mir an einem regnerischen Spätsommertag, während ich mein Zimmer aufräumte und feststellte, dass meine Mathematikbücher kaum nach einer anderen Kategorie einzuteilen waren.

Die eindimensionale Sortierung, die auf einem Vergleich von Elementen basiert, funktioniert nicht – die Frage, ob Olivgrün größer als Karmesinrot ist, ergibt keinen Sinn. Zunächst versuchte ich, die Farben numerisch einzuteilen: nach ihrer Helligkeit oder ihrem Bunton. Beide Methoden erzielten nicht das gewünschte Ergebnis: So könnte im ersten Fall ein Rot direkt zwischen zwei subtil unterschiedlichen Grüntönen stehen, während im zweiten Fall ein Pastellblau zwischen Laubgrün und Bordeauxviolett (zwei sehr dunkle Farben) eingeordnet wurde.

Mit dem Ziel, das Regal dennoch farblich ästhetisch zu sortieren, was (für mich) bedeutet, Farbkontraste zwischen nebeneinanderstehenden Büchern zu minimieren, war die Idee für ein Projekt geboren – die vergleichsbasierte, d.h. eindimensi-

onale Sortierung auf mehrdimensionale Objekte wie Farben zu erweitern und zu generalisieren. In diesem Projekt gehe ich den Fragestellungen nach, inwiefern mehrdimensionale Sortierung möglich und effizient lösbar ist, welche Rolle dabei die Graphentheorie spielt und welche Anwendungen sie neben dem (unter dem Gesichtspunkt der Kontrastminimierung) ästhetischsten Bücherregal hat (siehe [Abb. 1](#)).

Dabei gehe ich zu Beginn meiner Arbeit zunächst auf die theoretischen Grundlagen der mehrdimensionalen Sortierung ein ([Kap. 2.1](#)), beweise, dass es sich bei der angestrebten um eine Generalisierung der bekannten eindimensionalen Sortierung handelt ([Kap. 2.2](#)) und erläutere die äquivalente graphentheoretische Darstellung des Problems ([Kap. 3](#)). Nach der Präsentation der algorithmischen Komplexität ([Kap. 4.1](#)) und der implementierten Algorithmen stelle ich im Anschluss meine Entwicklung einer interaktiven Webanwendung vor ([Kap. 5](#)), in der man selbst verschiedene n -dimensionale Objekte sortieren kann – von abstrakten Vektoren ([Kap. 5.4](#)) über geografische Punkte ([Kap. 5.5](#)) bis hin zu visuellen Farben ([Kap. 5.6](#)). Diese Webanwendung visualisiert zudem die Funktionsweise der implementierten Algorithmen anschaulich und ermöglicht die praktische Sortierung auch größerer Datenmengen.

2. Beweisführung

In diesem Abschnitt soll bewiesen werden, dass es sich bei der im Folgenden definierten Kettensortierung um eine Generalisierung der vergleichsbasierten Sortierung auf mehrere Dimensionen handelt. Dabei werden zunächst auf der Zermelo-Fraenkel-Mengenlehre ([\[39\]](#), [\[92\]](#)) beruhende Definitionen formuliert und im zweiten Teil wird mittels vollständiger Induktion [\[72\]](#) ein Beweis aufgestellt. Dieser Beweis liefert die Grundlage für die anschließende Ausarbeitung, da sich die anschließend ausgearbeiteten Verfahren ohne ihn nicht als n -dimensionale Sortierung, sondern nur als beliebigen andere Algorithmus auf Listen von Vektoren bezeichnen könnten.

2.1 Definitionen

Eine Liste der Länge $n \in \mathbb{N}$ ist zum Zwecke dieser Arbeit eine injektive Abbildung l , die als Eingabe eine natürliche Zahl i (den Index) im Definitionsbereich $\{1, \dots, n\}$, folglich als



a)



b)

Abb. 1: Meine Mathematikbücher: links lexikographisch nach Nachname der Autor*innen, rechts nach perzeptueller Farbe mittels Simulated Annealing (Kap. 4.4.5) sortiert



\mathbb{D} bezeichnet, erhält und ein Element der Eingabemenge U zurückgibt. Der Wert von l an der Stelle i wird als l_i notiert. Der Teilabschnitt von p bis q , $p, q \in \mathbb{D}, p \leq q$ beschreibt die $(q - p + 1)$ -lange Liste $[l_p, l_{p+1}, \dots, l_{q-1}, l_q]$ und wird als $l_{p,q}$ notiert. Die Bildmenge Y meint die ungeordnete Menge aller in l vorkommenden Elemente, durch Injektivität gilt $|Y| = |l|$.

Das Vertauschen „Swap“ meint:

$$\text{Swap}(l, m, n)_i := \begin{cases} l_n & \text{falls } i=m \\ l_m & \text{falls } i=n \\ l_i & \text{sonst} \end{cases}$$

Die Menge der Permutationen „Perm“ einer Liste l wird definiert als:

$$\text{Perm}(l) := \{l' \mid l'_i = l_{\sigma(i)}, \sigma : \mathbb{D} \rightarrow \mathbb{D} \text{ bijektiv}\} \quad (1)$$

Eine Liste l , deren Zielmenge Teil einer strikten Totalordnung $(T, >)$ mit Ordnungsrelation $(>)$ ist, heißt genau dann sortiert, wenn gilt:

$$\forall i \in \mathbb{D} \setminus \{n\} : l_{i+1} > l_i \quad (2)$$

Die Kettenlänge $d_{\mathbb{L}}(l)$ einer Liste l , deren Zielmenge Teil eines metrischen Raums (T, d) mit Distanzfunktion d ist, wird definiert durch:

$$d_{\mathbb{L}}(l) := \sum_{i=1}^{n-1} d(l_i, l_{i+1}) \quad (3)$$

Eine solche Liste wird als kettensortiert bezeichnet, wenn gilt:

$$d_{\mathbb{L}}(l) = \min_{l' \in \text{Perm}(l)} d_{\mathbb{L}}(l') \quad (4)$$

2.2 Beweis: Jede sortierte Liste ist kettensortiert

Neben der abbildenden Definition der Liste kann eine Liste reeller Zahlen (also $U \subseteq \mathbb{R}$) äquivalent auch induktiv defi-

niert werden. Hierbei repräsentiere \mathbb{L} die Menge aller solcher Listen, ε die leere Liste mit Länge 0. \oplus bezeichne die strukturelle Konkatenation zweier Listen bzw. einer Liste und einem Element, welches als einelementige Liste interpretiert wird. Die Definition erfolgt wie folgt:

$$\varepsilon \in \mathbb{L} \quad (5)$$

$$l \in \mathbb{L} \wedge n = |l| \wedge e \in U \wedge \forall i \in [1, n] : e > l_i \Rightarrow \forall l^e \in \text{Ins}(l, e) : l^e \in \mathbb{L} \quad (6)$$

Dabei wird die Einfügemenge „Ins“ definiert als:

$$\text{Ins}(l, e) := \{l^e \mid n = |l|, i \in [1, n+1], l^e = l_{1:i-1} \oplus e \oplus l_{i:n}\} \quad (7)$$

Es gibt keine weiteren Listen.

Die Menge der sortierten Listen \mathbb{L}_{sort} bezeichnet dabei:

$$\varepsilon \in \mathbb{L}_{\text{sort}} \quad (8)$$

$$l \in \mathbb{L}_{\text{sort}} \wedge n = |l| \wedge e \in U \wedge \forall i \in [1, n] : e > l_i \Leftrightarrow l \oplus e \in \mathbb{L}_{\text{sort}} \quad (9)$$

($e > l_n$ wäre an dieser Stelle äquivalent, jedoch wird hier, um die Definitionen analog zu halten, die längere Variante gewählt.)

Dabei gilt: $\mathbb{L}_{\text{sort}} \subset \mathbb{L}$, da es sich bei der Konstruktion von \mathbb{L}_{sort} um einen Spezialfall von \mathbb{L} handelt, bei der $\text{Ins}(l, e) := \{l \oplus e\}$.

Nebensatz. Jede Liste mit total geordneter Zielmenge nach Kap. 2.1 ist Element von \mathbb{L} .

Beweis. Man betrachte eine abbildende Liste l . Aus dieser konstruiere man nun die L -Liste, w genannt, iterativ. So beginne man mit der leeren Liste ε (nach (5) $\varepsilon \in \mathbb{L}$) und betrachte stets das kleinste nicht betrachtete Element e zusammen mit seinem Index i . Existiert in l ein $j < i$, sodass $l_j < l_i$, so ist l_j bereits in w und e wird am darauffolgenden Index ein-

gesetzt, ansonsten am Index 0. Da stets $\forall i \in \mathbb{D}_w: e > w_i$ (sofern w noch nicht i enthält) und e an einer Stelle eingefügt (Ins) wird, ist (6) erfüllt und w eine Liste; da durch eindeutige Zuordnung alle Elemente enthalten sind und die Reihenfolge beibehalten wurde, teilen w und l alle Eigenschaften und sind damit identisch. ■

Somit ist die Konstruierbarkeit aus (5) und (6) als Eigenschaft aller Listen festzuhalten.

Satz. Jede sortierte Liste reeller Zahlen ist unter der Betragsmetrik kettersortiert.

Beweis. Die Definition anwendend, bedeutet dies:

$$\forall l \in \mathbb{L}_{\text{sort}} : d_{\mathbb{L}}(l) = \min_{l' \in \text{Perm}(l)} d_{\mathbb{L}}(l') \quad (10)$$

Da das Minimum einer Menge das Element bedeutet, für das kein kleineres Element existiert, ist eine gleichwertige Formulierung, dass für keine sortierte Liste eine Permutation dieser Liste existiert, die eine kleinere Kettenlänge hat. Dabei ist die Metrik $d(a, b) = |a - b|$.

Induktionsbeginn:

Länge 0: Die einzige Liste der Länge 0 ist ε , somit ist der Definitionsbereich $\{\}$. Nach (8) ist $\varepsilon \in \mathbb{L}_{\text{sort}}$. Somit kann keine Funktion eine Änderung der Elemente vornehmen (da keine solchen existieren), und jede sortierte Liste der Länge 0 ist sortiert und kettersortiert.

Länge 1: Eine sortierte Liste l der Länge 1 besteht aus einem Element, also $l = l_1$. Der Definitionsbereich ist $\{1\}$, die einzige Permutationsfunktion σ ist $\{(1,1)\}$. (Der Kürze halber werden Funktionen in diesem Abschnitt ihrer Definition zufolge als Mengen notiert.) Da $\forall i: \sigma(i) = i$ (Identität), wird keine Änderung der Elemente vorgenommen. Somit existiert keine Liste mit kürzerer Kettenlänge und jede sortierte Liste der Länge 1 ist kettersortiert.

Länge 2: Jede sortierte Liste l der Länge 2 erfüllt $l = l_1 \oplus l_2$, wobei $l_2 > l_1$. Die möglichen Permutationsfunktionen sind $\{(1,1), (2,2)\}, \{(1,2), (2,1)\}$. Beide ändern die Kettenstanz nicht, da die erste hier aufgeführte die Identität ist und die zweite die Liste umkehrt, was durch Kommutativität der Addition sowie Symmetrie der Metrik die Kettenstanz nicht ändert. Also ist auch jede sortierte Liste der Länge 2 kettersortiert.

Induktionsschritt:

l sei eine sortierte und kettersortierte Liste der Länge $n \in \mathbb{N}, n > 2$. Gemäß der induktiven Definition einer Liste wird nun ein neues Element $e \in U, e > l_n$ an einer beliebigen Position i in die Liste, welche fortan l^e genannt werde, eingefügt. Definitionsgemäß bleibt die Liste nur dann sor-

tiert, wenn $i = n + 1$, andernfalls wäre $l_{i+1} < l_i$ und die Liste unsortiert. Es wird nun bewiesen, dass beim Anfügen eines neuen Elements an genau dieser Stelle die Liste kettersortiert bleibt.

Zunächst wird die Kettenlänge betrachtet, die sich ergibt, falls $i = n + 1$. In diesem Fall gilt:

$$d_{\mathbb{L}}(l^e) = d_{\mathbb{L}}(l) + d(l_n, e)$$

Die Differenz zwischen bisheriger und neuer Kettenlänge

$$d_{\mathbb{L}}(l^e) - d_{\mathbb{L}}(l) = d(l_n, e)$$

wird als Δ_{opt} bezeichnet.

K sei nun die Menge aller Distanzen zwischen e und einem Element l_i an Index i der Liste. $e > l_i$ ist (wie auch $e > l_n$) gegeben. Da $i < n$, folgt (durch Definition von \mathbb{L}_{sort} und Transitivität von), dass $l_i < l_n < e$. Deshalb gilt $d(l_n, e) = e - l_n$ und $d(l_i, e) = e - l_i$, somit $l_i < l_n \Leftrightarrow e - l_i > e - l_n$. Da i beliebig gewählt wurde, ist die Distanz zwischen jedem Element und e größer als die zwischen l_n und e . Diese Distanz $d(l_n, e) = \Delta_{\text{opt}}$ ist folglich von allen Distanzen, die e involvieren, minimal.

Nun wird jede Permutation von l betrachtet und l' genannt. Da l kettersortiert ist, gilt:

$$d_{\mathbb{L}}(l) \leq d_{\mathbb{L}}(l')$$

Auf Basis dieser Permutation wird eine neue Liste l^e konstruiert, in die das Element e an einer Stelle i eingefügt wurde, also

$$l^e := l'_{1:i-1} \oplus e \oplus l'_{i:n}$$

$$\Delta := d_{\mathbb{L}}(l^e) - d_{\mathbb{L}}(l') \quad (11)$$

Es soll gezeigt werden, dass für jede Permutation l' gilt:

$$\Delta \geq \Delta_{\text{opt}} \quad (12)$$

So dass durch $d_{\mathbb{L}}(l') \geq d_{\mathbb{L}}(l)$ die Kettenstanz

$$d_{\mathbb{L}}(l^e) = d_{\mathbb{L}}(l') + \Delta \geq d_{\mathbb{L}}(l) + \Delta_{\text{opt}} = d_{\mathbb{L}}(l^e)$$

ist und somit $d_{\mathbb{L}}(l^e)$ tatsächlich die optimale Kettenstanz ist, wodurch l^e kettersortiert wäre.

Es wird eine Fallunterscheidung zwischen solchen Permutationen gemacht, bei denen e am Rand (Index $i = 1 \vee i = n + 1$) zu finden ist, und jenen, bei denen es zwischen zwei anderen Elementen ($i \in [2, n]$) vorliegt.

Fall 1. $i = 1 \vee i = n + 1$

Da e am Rand eingefügt wird, ist

$$d_{\mathbb{L}}(l^e) = d_{\mathbb{L}}(l') + d(l_i, e)$$

für ein beliebiges i aus der Indexmenge von l . Die Distanz ist dabei ein Element von K , da es eine Distanz zwischen e und einem Element von l ist. Da Δ_{opt} das minimale Element aus K ist, kann $\Delta = d(l_i, e)$ nicht geringer sein, sodass Ungleichung (12) zutrifft.

Fall 2. $i \in [2, n]$

Hierbei liegt e zwischen zwei Elementen l'_{i-1} und l'_i . Die neue Kettenlänge setzt sich nun zusammen aus der alten Kettenlänge minus der Distanz dieser beiden Elemente plus der Distanz jedes dieser Elemente mit dem neu eingefügten:

$$\Delta = d(l'_{i-1}, e) + d(l'_i, e) - d(l'_{i-1}, l'_i)$$

$$d_{\mathbb{L}}(l^e) = d_{\mathbb{L}}(l') + \Delta \quad (13)$$

Gegeben sind $l'_{i-1} < e$ und $l'_i < e$; o. B. d. A. wird nun von

$l'_{i-1} < l'_i < e$ ausgegangen, der Beweis kann analog durch Tauschen der beiden Elemente in der Ungleichung geführt werden. Explizit wird hier auf Eigenschaften der Betragsfunktion zurückgegriffen und mit

$$l'_{i-1} < l'_i < e$$

gilt:

$$\begin{aligned} \Delta &= |l'_{i-1} - e| + |l'_i - e| - |l'_{i-1} - l'_i| \\ &= (e - l'_{i-1}) + (e - l'_i) - (l'_i - l'_{i-1}) \\ &= e - l'_{i-1} + e - l'_i - l'_i + l'_{i-1} \\ &= e + e - l'_i - l'_i - l'_{i-1} + l'_{i-1} \\ &= 2 \cdot e - 2 \cdot l'_i \\ &= 2 \cdot (e - l'_i) \\ &= 2 \cdot |l'_i - e| \\ &= 2 \cdot d(l'_i, e) \end{aligned} \quad (14)$$

Δ ist folglich das Doppelte von $d(l'_i, e)$. Dabei handelt es sich um ein Element aus K , sodass Ungleichung (12) erfüllt ist.

Da keine Permutation der finalen Liste eine niedrigere Kettenlänge als die von l^e beim Anfügen des höchsten Elements erzielen kann, ist l^e – und damit jede sortierte Liste – ketten-sortiert. ■

2.3 Generalisierung auf n-dimensionale Vektorräume

Nun wurde bewiesen, dass die Sortierung einer Liste eindimensionaler Objekte, in diesem Fall repräsentiert durch reelle Zahlen, auch einer Kettensortierung dieser entspricht. Genauer: Jede sortierte Liste ist ketten-sortiert, jedoch nicht

zwangsläufig umgekehrt (aufgrund von Symmetrie der Kettenlänge im Gegensatz zur Sortierung).

Da die vergleichsbasierte Sortierung nur auf Listen von Elementen einer geordneten Menge ausführbar ist, die distanzbasierte Sortierung jedoch in jedem metrischen Raum angewendet werden kann und die Betragsmetrik der reellen Zahlen nur ein Sonderfall jeder durch eine p -Norm induzierten Metrik [36] n -dimensionaler Vektorräume ist [52], handelt es sich bei der Kettensortierung um eine Generalisierung der vergleichsbasierten Sortierung in Bezug auf die Dimensionalität der Eingaben.

Somit kann nun im Folgenden die Eigenschaft ‚sortiert‘ auch auf Listen mehrdimensionaler Objekte angewandt werden, da sie bis auf Umkehr der Liste die gleiche Bedeutung wie ‚ketten-sortiert‘ hat. Als Metrik wird sich aufgrund ihrer Generalisierungsfähigkeit von nun an auf eine durch eine p -Norm induzierte Metrik beschränkt, o.B.d.A. wird für die folgenden Beispiele die euklidische Metrik (gegeben durch $p = 2$) gewählt.

3. Graphentheoretische Grundlagen

3.1 Definitionen

Ein Graph G ist zum Zwecke dieser Arbeit ein Paar (V, E) , $E \subseteq V^2$ mit Knotenmenge V und Kantenmenge E . Ist $E = V^2$, so wird er als vollständig bezeichnet. Zusammen mit einer Kantengewichtsfunktion $d: E \mapsto \mathbb{R}$ gilt der Graph als kantengewichtet (kurz gewichtet), der Wert dieser Funktion für eine Kante ist ihr Gewicht.

Ein Weg ist eine Sequenz $p = v_1 v_2 \dots v_n$ paarweise verschiedener Knoten. Ein Weg ist ein Pfad, wenn gilt: $\forall i \in [1, n-1]: (p_i, p_{i+1}) \in E$. Das Gewicht eines solchen Pfades bezeichnet die Summe der Gewichte aller verbindenden Kanten:

$$d_p = \sum_{i=0}^{n-1} d(p_i, p_{i+1})$$

Ein Pfad wird Hamilton-Pfad genannt, sofern $n = |V|$ zutrifft, der Pfad also alle Knoten erreicht. Ein Hamilton-Pfad ist minimal, sofern kein Hamilton-Pfad des Graphen mit niedrigerem Gewicht existiert.

3.2 Anwendungen auf die Fragestellung

Um die Kettensortierung einer Liste l mit Eingabemenge $U \subseteq \mathbb{R}^n$, $n \in \mathbb{N}$ als graphentheoretisches Problem aufzufassen, betrachte man zunächst deren Bildmenge Y . Für diese konstruiere man nun den gewichteten Distanzgraphen $G = (V, E) = (Y, Y^2)$ mit Kantengewichtsfunktion $d(\vec{a}, \vec{b}) = \|\vec{a} - \vec{b}\|$. Der Graph ist vollständig.

Innerhalb dieses Graphen ist die Liste ein Hamilton-Pfad, denn zwischen jedem Paar aufeinanderfolgender Elemente existiert eine Kante (durch Vollständigkeit) und die Liste enthält alle Knoten (durch Konstruktion aus Bildmenge und Eindeutigkeit der Elemente der Liste). Die Kettendistanz der Liste ist gleich dem Gewicht dieses Pfads.

Da sich die Bildmenge durch Vertauschen von Elementen nicht ändert, ist auch jede Permutation der Liste ein valider Hamilton-Pfad, dessen Gewicht gleich der Kettenlänge der Liste ist ($d_L(p) := d_p$). Aus der Definition der Kettensortiertheit folgt, dass eine Liste genau dann kettensortiert ist, wenn ihr zugehöriger Hamilton-Pfad im Graphen der Bildmenge minimal ist. Somit kann mithilfe eines Algorithmus, der für einen Graphen dessen minimalen Hamilton-Pfad ermitteln kann, eine mehrdimensionale Liste sortiert werden.

4. Algorithmen

Für den Fall der eindimensionalen Sortierung existieren zahlreiche vergleichsbasierte Sortieralgorithmen, die sich in Eigenschaften wie asymptotischen Komplexitäten von Raum und Zeit, Stabilität und Vorgehensweise unterscheiden [3]. (Da die Elemente in einer Liste nicht mehrmals vorkommen können, spielt Stabilität für die Listen dieser Arbeit keine Rolle. Um Listen mit doppelten Werten dennoch sortieren zu können, sortiere man die deduplizierte Liste und füge die entfernten Werte an den Stellen nach dem gleichwertigen Element ein.) Bubble Sort, Insertion Sort und Selection Sort gehören zu den simpleren Verfahren, die jedoch aufgrund ihrer höheren Komplexität ineffizienter arbeiten; Merge Sort, Quick Sort [44] und Heap Sort dagegen werden auch in der Praxis genutzt [58], teils mit praktischen Anpassungen wie bei Timsort [8] (Python) oder Pattern-Defeating Quicksort [68] (Rust). Zwei Eigenschaften, die diese Sortieralgorithmen definieren, sind Monotonizität der Ausgabe (im eindimensionalen Fall gleich der in Kap. 2.1 definierten Sortiertheit) und Invarianz der Zielmenge.

Nun könnte ein mehrdimensionaler Sortieralgorithmus definiert werden als einer, der eine Liste als Eingabe erhält und eine kettensortierte Liste gleicher Elemente zurückgibt.

In diesem Abschnitt wird zunächst erläutert, warum eine derartige Definition sich für das Problem dieser Arbeit eher nicht eignet und stattdessen in zwei das Problem im allgemeinen Fall nicht lösende, aber dennoch in der Praxis sehr nützliche Arten von Algorithmen aufgeteilt werden muss. Im Anschluss werden Verfahren genannt, die ich zum Zweck dieser Arbeit auch in Rust [45] implementiert habe.

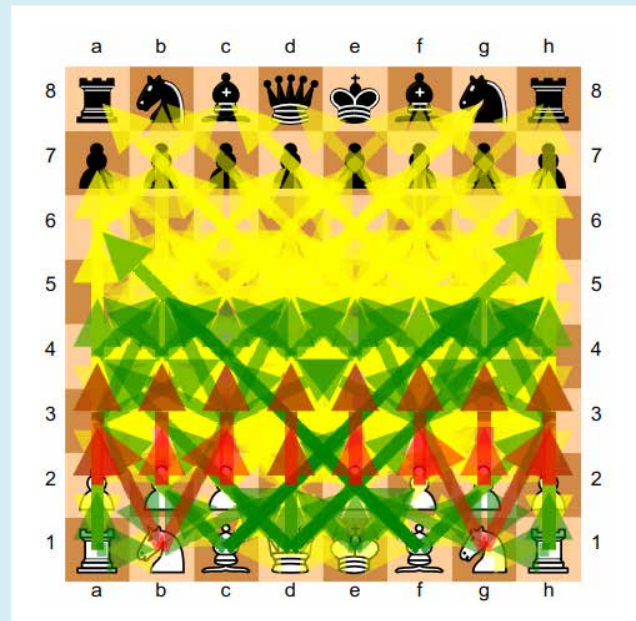
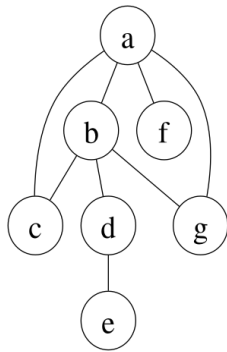
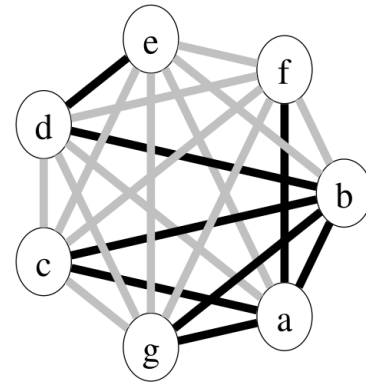


Abb. 2: Auch die Spielbaumtraversierung bis Tiefe n ist im Schach hartnäckig, da exponentiell [13] (S. 6–8).

4.1 Komplexität

Da die tatsächlich messbare Laufzeit eines Algorithmus von zu vielen algorithmisch irrelevanten Faktoren wie Hardware, Eingabestruktur und Ressourcenverfügbarkeit abhängt, wird in der theoretischen Informatik eine andere Methode gewählt, um Algorithmen bezüglich ihres Zeitverbrauchs nur abhängig von der Eingabelänge vergleichen zu können: die Landau-Symbole [10], [73], [88] oder auch O-Notation [28], [57], [85]. Diese bezeichnen das asymptotische Verhalten einer Funktion für beliebig größer werdende Eingaben. Wenn die Anzahl der Schritte, die ein Algorithmus für eine Liste der Eingabelänge n ausführt, nun durch eine Funktion f beschrieben werden kann, so liegt der Algorithmus in O , falls f asymptotisch nicht schneller wächst als g , und in $\Theta(g)$, falls f asymptotisch genauso schnell wächst wie g .

Die zu Beginn dieses Abschnitts genannten vergleichsbasierten Sortieralgorithmen wurden dabei nach ihrer Zeitkomplexität sortiert: Die als ‚simpler‘ bezeichneten Algorithmen liegen in $O(n^2)$, während die ‚praktischeren‘ in $O(n \cdot \log(n))$ liegen. Erstere werden daher als quadratisch, letztere als linearithmisch (ein Kofferwort aus ‚linear‘ und ‚logarithmisch‘) bezeichnet. Entscheidend ist im Folgenden allerdings eine andere Unterscheidung, die sich nur darauf bezieht, ob g durch ein Polynom beschreibbar ist oder nicht: polynomiell oder nicht-polynomiell. Erstere werden als in P und effizient beschrieben, während letztere ineffizient sind.

Abb. 3: Ein Beispiel für den Graphen G Abb. 4: Der für G neu erstellte Graph G' . Schwarze Kanten sind in E , graue nicht.

In der theoretischen Informatik werden die Komplexitätsklassen als Mengen dargestellt, die Entscheidungsprobleme beinhalten. \mathbf{P} ist die Menge aller Probleme, die sich durch eine deterministische Turingmaschine in einer polynomiellen Anzahl von Schritten („effizient“) lösen lassen, während \mathbf{NP} die umfasst, die eine nichtdeterministische Turingmaschine effizient lösen kann. Da jedes in \mathbf{P} lösbare Problem auch in \mathbf{NP} lösbar ist, gilt $\mathbf{P} \subseteq \mathbf{NP}$. (Obwohl in der praktischen Anwendung davon ausgegangen wird, dass $\mathbf{P} \neq \mathbf{NP}$ [53] (S. 465), bleibt dies ein ungelöstes Problem der Informatik [84]). Dagegen müssen \mathbf{NP} -schwere Probleme nicht in \mathbf{NP} liegen, sondern die ‚Schwierigkeit‘ dieser Klasse nur übertreffen.

4.1.1 Effiziente Lösbarkeit und Hartnäckigkeit

Probleme, die in \mathbf{P} liegen, werden als effizient lösbar, solche, die unter Annahme von $\mathbf{P} \neq \mathbf{NP}$ außerhalb von \mathbf{P} liegen, als hartnäckig bezeichnet (s. Abb. 2). Der Grund liegt in der unterschiedlichen Steigung der Terme für wachsende n : bei einem Polynom wird stets ein von n abhängiger Term addiert, bei einer Exponentialfunktion (die häufigste Form eines nicht-polynomiellen Terms) multipliziert.

4.1.2 Beweis der Hartnäckigkeit durch Reduktion

Um zu beweisen, dass ein Problem \mathbf{P}_1 hartnäckig ist, wird in der theoretischen Informatik eine Reduktion eines bekanntermaßen \mathbf{NP} -schweren Problems \mathbf{P}_0 auf das Problem \mathbf{P}_1 durchgeführt [53] (S. 452–454). Man zeigt, dass sich jede Instanz von \mathbf{P}_0 in polynomieller Zeit in eine Instanz von \mathbf{P}_1 transformieren lässt. Im Folgenden wird das Problem des minimalen Hamilton-Pfads im Fall vollständiger Graphen auf das bewiesenermaßen \mathbf{NP} -schwere Hamilton-Pfad-Problem [53] (S. 474–479) [38], [43] reduziert, welches aus der Frage besteht, ob zu einem beliebigen gegebenen Graphen ein Hamilton-Pfad existiert.

Dazu wird die Hypothese aufgestellt, es gäbe einen Algorithmus A , der das Problem für jeden vollständigen Graphen in polynomieller Zeit abhängig von der Anzahl der Knoten lösen könnte. Man betrachte nun jeden möglichen (insbesondere unvollständigen) Graphen $G = (V, E)$ (s. Abb. 3). Man konstruiere nun einen neuen vollständigen gewichteten Graphen (siehe Abb. 4) $G' = (V, E', d)$, wobei $E' = V^2$ und

$$d(a, b) = \begin{cases} 0 & \text{falls } (a, b) \in E \\ 1 & \text{falls } (a, b) \notin E \end{cases}$$

Nun führe man A mit der Eingabe G aus und erhalte den minimalen Hamilton-Pfad \mathbf{P}_{opt} .

Anhand dieser Ausgabe kann das Hamilton-Pfad-Problem für den Graphen G gelöst werden: ist das Gewicht von $\mathbf{P}_{\text{opt}} = 0$, so existiert für G ein Hamilton-Pfad, andernfalls nicht. Dies folgt daraus, dass A stets den minimalen Hamilton-Pfad auswählt – existiert einer im originalen Graphen, so kann dieser nur aus Kanten mit Gewicht 0 gebildet werden, sonst nicht.

Der Beweis basiert auf [53] (S. 479) und [51] und wurde hier statt auf das Travelling-Salesman-Problem auf das gegebene Problem des minimalen Hamilton-Pfads angewandt. Dies bestätigt erneut die Ähnlichkeit der beiden Probleme.

Somit entsteht ein Widerspruch: Die Aussagen, dass das Hamilton-Pfad-Problem \mathbf{NP} -schwer ist, und, dass der in \mathbf{P} liegende Algorithmus A es lösen kann, widersprechen sich. Somit ist die Hypothese falsch, A notwendigerweise ineffizient und das Problem \mathbf{NP} -schwer. Es wurde bisher nur gezeigt, dass das Problem des minimalen Hamilton-Pfads auf beliebigen Graphen \mathbf{NP} -schwer sein muss; für die im weiteren Verlauf dieser Arbeit betrachteten Distanzgraphen könnte das Problem immer noch in \mathbf{P} liegen. Ein Beweis der \mathbf{NP} -Schwere (hier ausgelassen) involviert die durch Translationsinvarianz von d gegebene metrische Universal konstruierbarkeit, die eine Reduktion ermöglicht.

4.2 Heuristik

Wie bereits in [Kap. 4.1.1](#) gezeigt wurde, kann die Kettensortierung im Allgemeinfall nicht in einer sinnvollen Zeitspanne gelöst werden. Aus diesem Grund werden auch andere Algorithmen betrachtet: Heuristiken. Eine Heuristik ist dabei ein effizienter Algorithmus, der ein hartnäckiges Problem nicht vollständig löst, sondern nur eine ungefähre Lösung liefert.

Im Fall der Sortierung mehrdimensionaler Objekte bedeutet dies, dass ein derartiger Algorithmus strategisch versucht, die Kettenlänge zu reduzieren, jedoch nicht zwangsläufig das globale Minimum (welches der Sortierung der Liste entspräche) findet.

Zudem ist eine weitere auf graphentheoretischer Grundlage beruhende Dichotomie der im Folgenden erläuterten Algorithmen sinnvoll: Konstruktionsalgorithmen auf der einen und Verbesserungsalgorithmen auf der anderen Seite. Dabei erhalten erstere als Eingabe eine Liste und geben einen Pfad zurück, während letztere versuchen, die Kettenlänge eines bestehenden Pfades zu verringern. In Bezug auf die verwendeten Datenstrukturen ist dies unerheblich, da auch Pfade als Listen repräsentiert werden; die Nutzerfreundlichkeit der Oberfläche nimmt allerdings zu, da Kombinationen des Ausführens von Algorithmen, die zu einer Annäherung an die Kettensortierung nicht beitragen, verhindert bzw. disinceniviert werden.

Ein Beispiel dafür ist, dass es keinen Sinn ergibt, nach dem Ausführen des Greedy-Algorithmus einen anderen Konstruktionsalgorithmus wie Nearest Neighbor zu nutzen, da dieser die vorherigen Ergebnisse nicht berücksichtigt. Diese somit zu limitieren (engl. *constrain*), wird als universeller Design-Grundsatz [\[55\]](#) verstanden.

Grundlegend betrachten Konstruktionsalgorithmen die Liste also eher als ungeordnete Menge, während sie bei Verbesserungsalgorithmen als geordneter Pfad interpretiert wird.

4.3 Pfadkonstruktion

Um die folgenden Konstruktionsalgorithmen anwenden zu können, wird eine Beispielliste l gewählt, welche sich aus sechs zweidimensionalen Vektoren zusammensetzt, folglich auch durch Farben im RGB-Farbraum dargestellt. Dabei repräsentieren die Komponenten des Vektors jeweils den Rot- und Blauwert der Farbe im Intervall $[0,1]$, der Grünwert wird auf 0 festgelegt:

$$l := \left[\begin{pmatrix} 0.6 \\ 0.2 \end{pmatrix}, \begin{pmatrix} 0.1 \\ 0.7 \end{pmatrix}, \begin{pmatrix} 0.9 \\ 0.4 \end{pmatrix}, \begin{pmatrix} 0.2 \\ 0.2 \end{pmatrix}, \begin{pmatrix} 0.8 \\ 0.7 \end{pmatrix}, \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix} \right] \triangleq \text{●} \text{●} \text{●} \text{●} \text{●} \text{●}$$

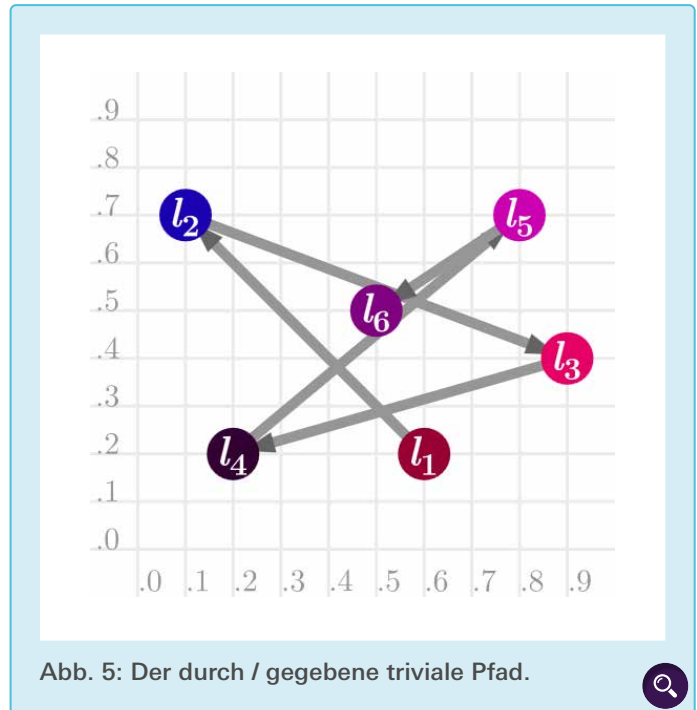


Abb. 5: Der durch / gegebene triviale Pfad.

Um einen Pfad sowie die darin enthaltenen Vektoren zu visualisieren, habe ich für die [Abb. 5](#) und [11–13](#) ein Programm mittels der JavaScript-Bibliothek p5.js [\[62\]](#) entwickelt, welches die Listenelemente und ihre Farben in einem zweidimensionalen kartesischen Koordinatensystem darstellt. Die genannten Abbildungen entspringen diesem.

4.3.1 Triviale Pfadkonstruktion

Die triviale Pfadkonstruktion gibt die Eingabeliste unverändert zurück – in diesem Fall entstünde der in [Abb. 5](#) erkennbare Pfad. Dieser Algorithmus ermöglicht es, beliebige Pfade selbst zu erstellen und zu verändern, ohne dabei an eine bestimmte Heuristik gebunden zu sein. Zudem können so die Verbesserungsalgorithmen teils besser dargestellt werden. Eine Alternative dieses Verfahrens ist, die Liste vorher zufällig zu mischen.

4.3.2 Brute Force

Der Brute-Force-Algorithmus kann tatsächlich eine Liste mehrdimensional sortieren, also das Minimum aller Permutationen finden, indem jede dieser Permutationen auf ihre Kettenlänge überprüft und jene mit der minimalen zurückgegeben wird (siehe [Abb. 6](#))

Die Zeitkomplexität dieses Algorithmus liegt in $O(n!)$, wächst also proportional zur Fakultät der Eingabelänge, da diese zugleich die Anzahl der Permutationen einer derartigen Liste beschreibt. In der Praxis zeigt sich, dass das Verfahren für Listen mit zehn oder weniger Elementen durchaus nutzbar ist, je-






#	p	$d_{\mathbb{L}}(p)$	min
1		3.43	3.43
2		3.07	3.07
...	(233 weitere)		
236		1.88	1.88
...	(483 weitere)		
718		3.07	1.88
719		3.43	1.88

Abb. 6: Brute Force überprüft jede Permutation





































Pfad p ($d_{\mathbb{L}}(p)$)	Nachbarn von $p _p $
 (0)	 0.32  0.36  0.40  0.54  0.71
  (0.32)	 0.36  0.41  0.42  0.45
   (0.68)	 0.32  0.70  0.78
    (0.99)	 0.73  0.85
     (1.72)	 0.51
      (2.23)	

Abb. 7: Stets wird vom aktuellen Punkt aus der nächste Nachbar gesucht

doch ab einer Länge von 14 Elementen mit einem geschätzten Zeitaufwand von zwei Stunden keine Option mehr darstellt (siehe auch [Kap. 4.1.1](#))

4.3.3 Nächster Nachbar

Beim Nächster-Nachbar-Algorithmus (engl. *nearest neighbor*, kurz NN) handelt es sich um ein sog. naives gieriges Verfahren. Dabei wird vom ersten Punkt der Liste aus begonnen und stets der Punkt ausgewählt und folglich betrachtet, dessen Distanz zum aktuell betrachteten minimal ist und noch nicht im zu erstellenden Pfad enthalten ist, bis alle Punkte im Pfad enthalten sind (s. [Abb. 7](#)).

Da es stets einen konkreten Bezugspunkt gibt, von dem aus vorgegangen wird, ist der Algorithmus einfach zu verstehen und zu visualisieren. Für die praktische Anwendung reicht er nicht, da stets nur das nächste Element berücksichtigt und der restliche Kontext vernachlässigt wird – so kommt es insbesondere am Ende des Pfades meist zu besonders langen Kanten hin zu Knoten, die unter Reduzierung der Kettenlänge bereits vorher hätten besucht werden können, es allerdings nicht wurden, da andere Knoten unmittelbar näher lagen. Die Komplexität ist quadratisch, da für jeden hinzuzufügenden Knoten jeder andere Knoten überprüft wird.

Da stets beim ersten Punkt dieser Liste begonnen wird, ist der Algorithmus anordnungsabhängig. [Abb. 8](#) zeigt die verschiedenen erzeugten Pfade im Fall von l (14) – aufgrund des unidirektionalen Auswahlverfahrens weisen sie alle eine unterschiedliche Kettenlänge auf.

Um den NN-Algorithmus unabhängig vom ersten Element der Liste zu gestalten, kann auf Kosten der nun kubischen Laufzeitkomplexität der bisherige Algorithmus für alle Rotationen der Originalliste ausgeführt werden und das Ergebnis geringster Kettenlänge ausgewählt werden. Eine alternative kubische Generalisierung ist die Aufhebung der Beschränkung des Anfügens am Ende auf eine optimale Einfügetaktik an einer beliebigen Stelle.

4.3.4 Greedy

Der Greedy-Algorithmus (auch Multiple Fragments [\[49\]](#)) enumeriert alle Kanten und sortiert sie nach ihrem Gewicht aufsteigend. Diese werden in einem Stapel abgelegt und stets wird die erste Kante ausgewählt, entfernt sowie dem Pfad angefügt, die die Validität nicht verletzt, also keine bereits zweifach verbundenen Knoten besucht oder Zyklen erstellt (s. [Abb. 9](#)).

Sobald die Länge dieser Liste von Kanten gleich der der Eingabe minus eins ist, ist ein valider Pfad gefunden, da die Validität nicht verletzt wurde und alle Knoten besucht werden. Zudem existiert stets ein solcher Pfad, da von der Vollständigkeit des Graphen ausgegangen wird. Bei Wahl zweckmäßiger Datenstrukturen liegt die Zeitkomplexität in $O(n^2 \cdot \log(n))$.

Die Fragmentierung kann analog zu [Kap. 4.3.3](#) zum Schluss zu suboptimalen Kanten führen, jedoch wird das Problem der Nichtberücksichtigung von „Außenseitern“ zumeist umgangen, da auch der Weg von einem solchen zu einem beliebigen anderen Knoten zumeist kürzer ist als ein anderer, der zwischen „Clustern“ wechselt.











































p_0	p	$d_{\mathbb{L}}(p)$
	     	2.23
	     	2.22
	     	1.90
	     	2.25
	     	1.93
	     	2.20

Abb. 8: Die NN-Pfade abhängig vom Anfangspunkt

4.3.5 Ganzzahlige lineare Optimierung und Branch-and-Bound

Um die optimale Lösung zu finden, ohne jede Permutation überprüfen zu müssen, eignet sich die Neuformulierung des Problems in Begriffen der ganzzahligen linearen Optimierung (ILP). Für eine Instanz der Länge n , einen als Adjazenzmatrix A_{ij} repräsentierten Distanzgraphen $((V, E), d)$ und einen Pfad p repräsentierenden $\{0,1\}$ -wertigen Relationsmatrix $X_{ij} = (V_i, V_j) \in p$ habe ich das folgende ILP (naiv $n^2 + 3n + 1$ Nebenbedingungen) erarbeitet:

$$\min \sum_{i=1}^n \sum_{j=1}^n X_{ij} \cdot A_{ij}, \text{ s.t.}$$

$$\forall i \in [1, n] : \sum_{j=1}^n X_{ij} + X_{ji} \geq 1$$

$$\wedge \forall i \in [1, n] : \sum_{j=1}^n X_{ij} \leq 1$$

$$\wedge \forall j \in [1, n] : \sum_{i=1}^n X_{ij} \leq 1$$

$$\wedge \sum_{i=1}^n \sum_{j=1}^n X_{ij} = n - 1$$

$$\wedge \forall i \in [1, n] : \forall j \in [1, n] : X_{ij} + X_{ji} \leq 1$$

Jeder Eintrag der Matrix X ist eine Variable des ILP. Minimiert wird dabei das Gewicht des Pfads, unter den Bedingungen, dass jeder Knoten erreicht wird, in jeder Zeile und jeder Spalte der Matrix höchstens eine Kante Teil des Pfads ist, insgesamt $n - 1$ Kanten enthalten sind und keine Kante und ihre Gegenkante gewählt wird (dies verhindert auch Schlingen).

Eine alternative und TSP-ähnlichere Modellierung könnte (durch Symmetrie) nur Kanten oberhalb der Hauptdiagonale betrachten. So wäre das ILP effizienter (da kleiner), jedoch würden die ersten beiden Nebenbedingungen deutlich komplexer, da gültige Lösungen pro Zeile und Spalte 0 bis 2 Kanten auswählen könnten und eine neue Validitätseinschränkung formuliert werden muss. In meiner Arbeit verzichte ich auf diese Optimierung.

Eine zulässige Lösung [76] ist dabei nicht zwangsläufig eine valide Permutation, da Zyklen nicht betrachtet werden: Sie müssen nach Ermitteln einer invaliden Lösung als neue Nebenbedingungen hinzugefügt werden. Da der Beweis in Kap. 4.1.2 zutrifft, ist auch dieses ILP (bzw. dessen azyklische Variation) hartnäckig; allerdings können die konstanten Faktoren mit einem geeigneten Algorithmus stark gesenkt werden. Fokus meiner Arbeit ist dabei ein Branch-and-Bound-Verfahren [64]: Zunächst wird (mithilfe einer externen Bibliothek) das effizient lösbare bedingungsgleiche in X reellwertige LP gelöst (da statt diskreten Werten kontinuierliche ermittelt werden, handelt es sich dabei um eine Relaxierung), um die Menge möglicher Lösungen einzugrenzen; im Anschluss werden die kontinuierlichen Werte durch Aufspalten der Wertebereiche der Variablen fixiert. Das genaue Verfahren, z. B. in [50] und [83] für das symmetrische TSP oder in [60] für allgemeine ILP beschrieben, wird hier der Kürze halber ausgespart; jedoch schafft es meine Implementierung in der Praxis, Instanzen bis $n = 50$ in hinnehmbarer Zeit zu lösen.

4.4 Pfadverbesserung

Die Pfadverbesserungsalgorithmen werden im Folgenden am Beispiel des eben durch NN (bei Start am ersten Element der













						
	(0)	.71	.36	.40	.54	.32
	.71	(0)	.85	.51	.70	.45
	.36	.85	(0)	.73	.32	.41
	.40	.51	.73	(0)	.78	.42
	.54	.70	.32	.78	(0)	.36
	.32	.45	.41	.42	.36	(0)

Abb. 9: Adjazenzmatrix $A \in \mathbb{R}^{|P| \times |P|}$: $A_{ij} = d(d_i, d_j)$ des Graphen der Bildmenge von I

i	$e \in E$	$d(e_0, e_1)$	$e \in p?$
0	● → ●	0.32	ja
1	● → ●	0.36	ja
2	● → ●	0.32	ja
3	● → ●	0.73	ja
4	● → ●	0.51	ja
5	● → ●	0.71	nein

Abb. 10: Alle im Pfad vorkommenden Distanzen sowie die ausgelassene



Liste) erstellten Pfads,

$p = \text{●} \text{●} \text{●} \text{●} \text{●} \text{●}$

erklärt (siehe Abb. 11).

4.4.1 Rotation

Mit einer linearen Laufzeit ist die Rotation das algorithmisch einfachste Verfahren aus dieser Liste. Dabei wird das Maximum der Kantengewichte aller Kanten des Pfads sowie der nicht berücksichtigten Kante zwischen Start und Ziel des Pfads ermittelt. Falls diese ein höheres Gewicht als alle im Pfad enthaltenen aufweist, so ist der Pfad bereits rotations-optimal und kann nicht optimiert werden. Ansonsten wird die maximale Kante aus dem Pfad entfernt und die neue Kante hinzugefügt. Durch Erweitern auf sequenzielle Teilpfade können ästhetische lokale Minima gefunden werden.

Abb. 10 zeigt die Anwendung der Rotation auf den Beispielpfad. Hierbei zeigt sich, dass eine im Pfad vorkommende Kante (● → ●) ein höheres Gewicht als die ausgelassene (● → ●) aufweist. Somit kann durch eine Rotation der den Pfad repräsentierenden Liste die Kettenlänge (in diesem Fall jedoch nur um $0,73 - 0,71 = 0,02$ Einheiten) reduziert werden.

4.4.2 Swap

Swap betrachtet den Pfad als Liste und überprüft, ob das Tauschen („Swap“ nach Kap. 2.1) zweier Elemente zu einem Pfad mit geringerer Kettenlänge führt. Bei Konstruktion mittels einer nicht-trivialen Heuristik (wie auch hier) ist dies allerdings selten der Fall, weshalb das Verfahren eher nur als Grundlage für andere wie Kap. 4.4.3 und 4.4.5 dient.

4.4.3 Das 2-opt-Verfahren

Das 2-opt-Verfahren [29] beruht auf der folgenden geometrischen Erkenntnis: Sobald sich in der zweidimensionalen geometrischen Repräsentation eines Pfads zwei als Strecken repräsentierte Kanten schneiden, kann die Kettenlänge des Pfades optimiert werden, indem der Schnittpunkt wie in Abb. 12 durch Tauschen zweier Kanten entfernt wird. Diese Pfadmodifikation wird als 2-opt-Tausch bezeichnet.

Dabei wird die Kettenlänge stets reduziert, da die neue Strecke zwei Punkte direkt verbindet, statt einen ‚Umweg‘ zu enthalten. In Bezug auf einen Pfad als Liste bedeutet ein 2-opt-Tausch dabei, eine Teilliste dieser umzukehren. Durch Ungerichtetheit des Graphen ändert sich die Kettenlänge der Teilliste nicht.

Während die Existenz eines solchen Schnittpunkts im zweidimensionalen kartesischen Koordinatensystem mithilfe eines Sweep-line-Algorithmus in linearithmischer Laufzeit ermittelbar ist [9], existiert ein solches Verfahren für höhere Dimensionen nicht. Aus diesem Grund ist die Laufzeit im allgemeinen Fall pro Optimierungsschritt quadratisch, da jede Kante mit jeder weiteren Kante auf Tauschbarkeit überprüft wird, die auch nicht mehr nur auf Schnittpunkten basiert, sondern die Kettendistanzen vergleicht.

Während das Auflösen aller Überschneidungen im Pessimalfall eine Laufzeitkomplexität von $\tilde{O}(n^{10})$ (die Tilde signalisiert die Vernachlässigung logarithmischer Faktoren) [7] benötigt, konvergiert der Algorithmus meist schneller. Ist kein 2-opt-Tausch mehr ausführbar, wird der Pfad 2-opt-optimal genannt.

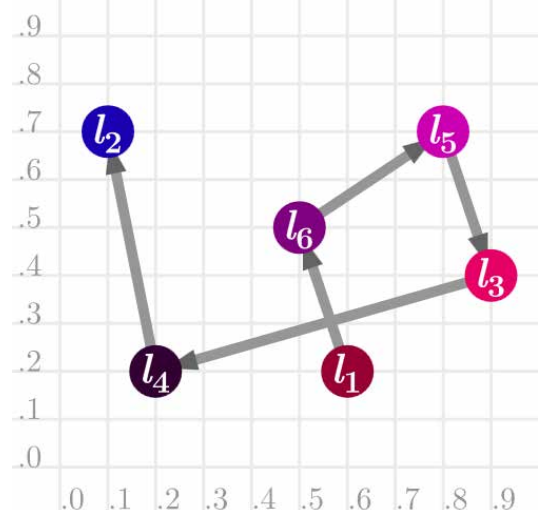


Abb. 11: Beispielpfad p



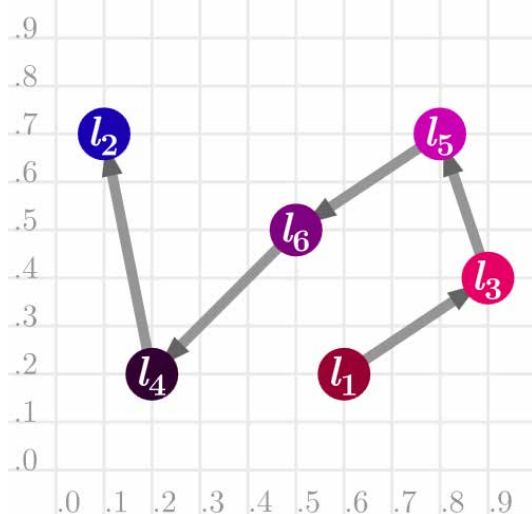


Abb. 12: Der 2-opt-optimale Pfad p' nach 2-opt-Tausch von l_3 und l_6

4.4.4 3-opt und k-opt Verfahren

Ähnlich funktioniert das 3-opt-Verfahren: Hierbei werden allerdings zwei statt drei Kanten getauscht. Da kein exaktes geometrisches Äquivalent existiert, wird die Liste betrachtet. Für einen 3-opt-Swap des Pfads p an den paarweise verschiedenen Indizes $i, j, k \in \mathbb{D}$ existieren dabei vier Möglichkeiten, die verschiedene Teillisten dabei umzukehren oder nicht, wenn die Permutationen der Identität und des 2-opt-Tausches vernachlässigt werden, sonst acht. 3-opt generalisiert im letzteren Fall 2-opt, auch eine Generalisierung auf beliebige $k \in \mathbb{N}, k \geq 2$ ist möglich [22]. Effizient gelöst werden kann das Problem jedoch durch beliebig ansteigende k nicht, denn die Auswahl aller möglichen Kanten liegt in $O(k!)$.

4.4.5 Simulated Annealing

Simulated Annealing ist ein Verfahren, welches auf der physikalischen Kristallisierung von Materialien beruht [54]. Dabei wird das Swap-Verfahren so erweitert, dass nicht nur solche Kommutationen ausgeführt werden, die kürzere Kettenlängen erzeugen, sondern zu Beginn auf Basis einer Zufallsvariable auch solche, die es nicht tun. Im Laufe der Zeit wird die Wahrscheinlichkeit dafür kontinuierlich reduziert, bis zum Schluss ein swap-optimaler Pfad gefunden wurde – da jedoch ein größerer Teil der möglichen Pfade abgedeckt werden kann, untertrifft er den des Swap-Verfahrens zumeist stark. Im Fall von p (siehe Abb. 13) konnte sogar der optimale Pfad gefunden werden.

Algorithmisch wird dabei eine Starttemperatur $t \in [0,1]$

festgelegt und ein Eingabepfad p erhalten. Man betrachte nun in einem Graphen mit

$$G = (\text{Perm}(p), \{(p', p'') \mid i, j \in \mathbb{D}, p' \in \text{Perm}(p), p'' := \text{Swap}(p', i, j)\})$$

die Nachbarn von p und wähle ein zufälliges p' aus, dies geschieht durch Wahl von i und $i \neq j$ und Tauschen der Elemente. Falls nun

$$d_{\mathbb{L}}(p') < d_{\mathbb{L}}(p)$$

wird mit p' in jedem Fall weiterverfahren, ansonsten beträgt die Wahrscheinlichkeit dafür

$$e^{-\frac{d_{\mathbb{L}}(p') - d_{\mathbb{L}}(p)}{t}}$$

sinkt also kontinuierlich mit der Temperatur. t wird nun arithmetisch um eine Konstante k verringert, je nach gewünschter Konvergenzgeschwindigkeit – für den Fall der Farbsortierung von $n \in [10, 50]$ hat sich der Bereich um $k \approx 10^{-10}$ bewährt, sodass 10 Millionen Iterationen stattfinden – mit $t \rightarrow 0$ findet kein Tausch mehr statt und der Algorithmus wird abgebrochen.

5. Webanwendung

Was nützen die besten Algorithmen, wenn sie nicht auf Daten aus der realen Welt angewandt werden können? Um dies und vieles mehr zu ermöglichen, habe ich eine interaktive Webanwendung entwickelt, die genutzt werden kann, um mehrdimensionale Daten verschiedener Kategorien zu sortieren. Im

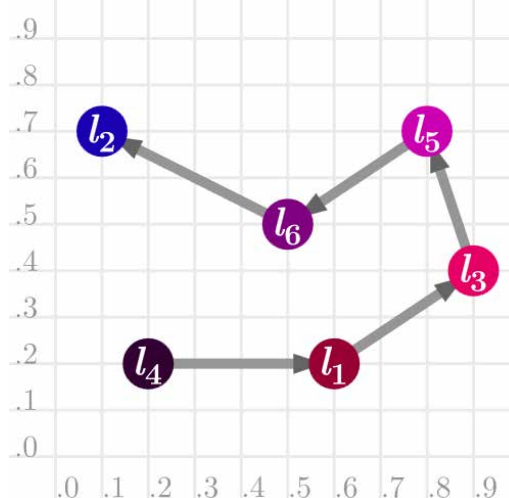


Abb. 13: Simulated Annealing

folgenden Abschnitt werden dabei zunächst die verwendeten Technologien grundlegend erklärt, bevor auf die einzelnen Unterseiten eingegangen wird. Dabei sticht der Abschnitt zur Sortierung von Farben hervor, da dieser meine ursprüngliche Fragestellung beantwortet: Wie können algorithmisch Bücher ästhetisch nach ihrer Farbe sortiert werden?

5.1 Grundlagen und Open-Source-Implementierung

Der Quelltext der gesamten Anwendung ist frei und unter der GNU GPL v3 [82] veröffentlicht. Über GitHub ist der des Frontends unter [16]; der des Backends unter [15] zu erreichen.

5.1.1 Frontend

Das Frontend der Anwendung habe ich in Svelte [40] und TypeScript [12], [42] geschrieben; es verwaltet mittels SvelteKit [41] die Unterseiten. Dabei erfolgt die Modellierung nach dem Component-Prinzip – individuelle Components verwalten dabei sowohl einen Zustand als auch die dazugehörigen (reaktiven) DOM-Elemente. So existiert etwa ein `PathProperties`-Component, welches die Eigenschaften eines Pfades (wie Länge, Kettenlänge und Kettensortiertheit) anzeigt und selbst akquiriert, und analog ein `PathAlgorithms`-Component, das das Ausführen von Konstruktions- und Verbesserungsalgorithmen über die Serververbindung ermöglicht.

5.1.2 Backend

Um maximale (zeitliche) Effizienz der Algorithmen sowie Speicher- und Typensicherheit zu gewährleisten, habe ich mich dafür entschieden, für das Backend der Anwendung die Programmiersprache Rust [45], [61] zu nutzen. Ein Pfadkonstruktionsalgorithmus ist dabei beispielsweise eine Funktion `fn(PathCreateContext) -> Path`, wobei `Path` ein Typ ist, der eine Liste als Pfad repräsentiert und `PathCreateContext` als `struct PathCreateContext {action: ActionContext, dim: u8, points: Points, metric: Metric }` definiert ist.

Der Code in Abb. 14 zeigt eine Implementierung von NN in Rust. Dabei wird an einen Pfad (wie in Kap. 4.3.3 beschrieben) stets der nächste nicht besuchte Nachbar angehängt. In der aktuellen Version wird dagegen stets die Adjazenzmatrix (siehe Abb. 9) vollständig berechnet. Dies sorgt zwar (bei Dimension d und Listenlänge n) für $\Omega(d \cdot n^2)$, doch auf die Distanz zweier Elemente in Konstantzeit zugreifen zu können ($d(a, b) \in O(1)$ statt $O(d)$) [31], [87], ist zumeist lauffechnisch vorteilhaft.

```
pub fn nearest_neighbor(ctx: PathCreateContext)
-> Path {
    let PathCreateContext {
        action, dim, points: values, norm
    } = ctx;

    let mut visited = HashSet::new();
    let mut path = Path::try_new(
        vec![values[0].clone()],
        dim).expect("invalid dimension");
    while path.len() != values.len() {
        let last = &path[path.len() - 1];
        visited.insert(last.clone());

        let min = values
            .iter()
            .filter(|&point|
                !visited.contains(point))
            .min_by_key(|point|
                point.comparable_dist(
                    last, norm
                ))
            .unwrap();

        path.push(min.clone());
        action.send(
            PathCreation::from_path(
                path.clone()
            ).progress(path.len() as f32
                / values.len() as f32),
        );
    }

    path
}
```

Abb. 14: NN als Rust-Programm. Alle anderen aus Kap. 4 implementierten Verfahren sind über GitHub (siehe Kap. 5.1) ersichtlich.

5.2 Server-Client-Kommunikation

In bisherigen Projekten (wie [14]) nutzte ich das HTTP-Protokoll, um mit dem Server zu kommunizieren. Dieses zeichnet sich durch ein Request-Response-Schema aus, bei dem auf eine Anfrage (Request) des Clients genau eine Antwort (Response) des Servers erfolgen soll, die idealerweise nur auf den Daten der Anfrage beruht.

Während einige Methoden entwickelt wurden, um diese Einschränkungen aufzuheben oder ihre negativen Auswirkungen zu mindern (dazu zählt das wiederholte Anfragen einer Ressource vom Server oder das in HTTP/2 implementierte Server-Push-Verfahren [93]; während erstere negative Auswirkungen auf die Performanz der Anwendung hat, ist zweite nicht universell nutzbar und keine Alternative für den gewählten Servertyp), eignet sich für diese derartig dynamische Anwendung, bei der eine einzige Anfrage hunderte Status-Antworten zur Folge haben sollte, um Responsivität zu gewährleisten, eher ein anderes in Webbrowsern mittlerweile universell implementiertes Protokoll: das WebSocket-Proto-

```

{"type":"action", "latency":100,
 "action":{"type":"createPath",
 "method":
 {"type":"nearestNeighbor"},
 "dimensions":3, "values":[[0.6,
 0.2], [0.1, 0.7], [0.9, 0.4],
 [0.2, 0.2], [0.5, 0.5], [0.8,
 0.7]]}}

{"type":"pathCreation",
 "currentEdges":[[[0.6, 0.2], [0.5,
 0.5]]], "progress":0.33333334}
{"type":"pathCreation",
 "currentEdges":[[[0.6, 0.2], [0.5,
 0.5]], [[0.5, 0.5], [0.8, 0.7]]],
 "progress":0.5}
{"type":"pathCreation",
 "currentEdges":[[[0.6, 0.2], [0.5,
 0.5]], [[0.5, 0.5], [0.8, 0.7]],
 [[0.8, 0.7], [0.9, 0.4]]],
 "progress":0.66666667}
...
{"type":"pathCreation",
 "donePath":[[0.6, 0.2], [0.5,
 0.5], [0.8, 0.7], [0.9, 0.4],
 [0.2, 0.2], [0.1, 0.7]],
 "currentEdges":[[[0.6, 0.2], [0.5,
 0.5]], [[0.5, 0.5], [0.8, 0.7]],
 [[0.8, 0.7], [0.9, 0.4]], [[0.9,
 0.4], [0.2, 0.2]], [[0.2, 0.2],
 [0.1, 0.7]]], "progress":1.0}

```

Abb. 15: Eine Anfrage (nach Kap. 4.3.3) und die fünf darauffolgenden Antworten.

koll [32]. Dieses baut auf einem dauerhaft aktiven TCP-Server auf und ermöglicht so bidirektionale zustandsbehaftete Kommunikation.

Server und Client tauschen JSON-Dokumente aus, die auf der Backend-Seite in Rust mithilfe von `serde` [86] und auf der Frontend-Seite in TypeScript mithilfe von `zod` [63] typensicher deserialisiert werden. Im Code in Abb. 15 wird gezeigt, wie eine solche Kommunikation aussehen kann: zunächst fragt der Client die Erstellung eines Pfads an und übergibt die gewünschte Konstruktionsmethode und Minimallatenz. (Falls der Server zu viele Antworten in zu kurzer Zeit verschickt, kann dies zu Unresponsivität des Clients und einem potenziellen Speicherleck führen – aus diesem Grund kann die Latenz, die der Server zwischen Abschicken zweier Antworten mindestens wartet, hier konfiguriert werden), daraufhin schickt der Server für jeden relevanten Schritt eine Antwort zurück, bis der vollständige Pfad ausgegeben wird. Im Fall von NN ist dies für die Responsivität noch nicht entscheidend; bei länger andauernden Prozessen wie mehrschrittigen Verbesserungsalgorithmen dagegen signifikant.

5.3 Zahlen sortieren

Auf der Seite `/sort-integers` können ganze Zahlen eingegeben und nach einem Sortierungsalgorithmus der Wahl (aktuell implementiert sind Bubble Sort, Insertion Sort, Selection Sort, Quick Sort und Merge Sort) aufsteigend sortiert werden. Dabei wird bei jedem durch den Server ausgeführten algorithmischen Schritt – etwa Vergleich zweier Werte, Vertauschen oder Einstufen als bereits sortiert – die Liste samt der aktuell ausgeführten Schritte zurückgegeben. Dabei kann die Liste neben der üblichen Darstellung ihrer Elemente auch in einem Balkendiagramm dargestellt werden. Abb. 16 zeigt ein solches Diagramm, das Quick Sort angewandt auf die ersten sechzehn Zahlen der OEIS-Sequenz A107833 [75], [79] darstellt.

5.4 Vektoren sortieren

Die Seite `/sort-vectors` ermöglicht die Sortierung einer Liste von Vektoren beliebiger Dimension. Nach Festlegen einer Dimension können Vektoren hinzugefügt, ihre Komponenten modifiziert und anschließend mittels `PathAlgorithms` ein Pfad erzeugt werden. Da jede andere Seite ebenfalls n -dimensionale Objekte sortiert, die als Vektoren repräsentiert werden können, enthalten sie eine Weiterleitung auf diese Seite, welche die Vektoren als URL-Parameter übergibt.

Um die Daten zu visualisieren, wurde ein kraftgerichteter (*force-directed*) Graph-Layout-Algorithmus implementiert. Dieser enthält ein Partikelsystem, sodass jeder Vektor durch ein Partikel repräsentiert wird, auf welches physikalische Kräfte wirken. Dabei existiert zwischen jedem Paar von Vektoren eine Feder mit einer Ruhelänge der durch die gewählte Metrik gegebenen Distanz, sodass analog zum Hook'schen Gesetz [23] Kräfte auf die betroffenen Partikel wirken. So approximiert die euklidische Distanz zwischen den sehbaren Punkten die genannte Distanz zwischen den Vektoren und skaliert damit mehrdimensional. Im zweidimensionalen euklidischen Fall konvergiert der Graph bis auf Rotation, Skalierung und Spiegelung dabei zu den tatsächlichen Punkten. Durch einen einstellbaren Parameter der Initialgeschwindigkeit kann zwischen einer zeitintensiveren, genaueren Darstellung und einer schnelleren und damit ungenaueren Konvergenz gewählt werden.

5.5 Orte sortieren

Die Probleme des bilokal sesshaften, polylokal handelnden Händlers, der drohnenbasierten Paketzustellung sowie der optimalen U-Bahn-Strecke werden durch die Seite `/sort-places` lösbar. Dabei zeigt eine interaktive auf der Basis von Leaflet [2] und OpenStreetMap [24] entwickelte Karte die Punkte sowie den Pfad an.

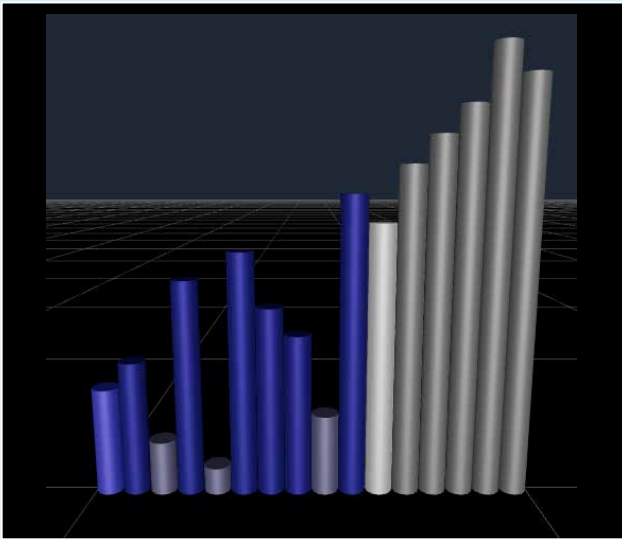


Abb. 16: Quick Sort partitioniert die Teilliste



Als Beispiel wurden für [Abb. 17](#) die Landeshauptstädte der Bundesländer Deutschlands ausgewählt, zwischen denen z. B. ein Paketdienst verkehren soll. Dazu wurden ihre geographischen Koordinaten auf der Website eingegeben und im Anschluss mittels ILP ([Kap. 4.3.5](#)) der kürzeste Pfad konstruiert. In diesem Fall beginnt die Strecke in Düsseldorf und endet in Dresden. Eine möglichst effizient entworfene Eisenbahnlinie mit dem Ziel, all diese Orte zu verbinden, sollte ebenfalls den genannten Start- und Zielpunkt nutzen.

5.6 Farben sortieren

Im Alltag gibt es viele Dinge, die nach Farben sortiert werden können – während Bücher zumeist alphabetisch oder nach Kategorien sortiert werden sollten, ist dies bei Malstiften jeglicher Art, farblicher Dekoration und womöglich auch Kleidung anders. Hier ergibt eine Farbsortierung Sinn, und die meisten Menschen haben eine intuitive Vorstellung davon, was das bedeutet: Ähnliche Farben gehören nah zueinander und unterschiedliche auseinander – es scheint also eine quantifizierbare intuitive Distanz zwischen zwei Farben zu geben. Während Ästhetik subjektiv bleibt und daher nicht die ästhetischste Liste für jeden existieren kann, treffe ich die Annahme, dass die Kettensortierung einer Liste am ästhetischsten ist, da sie Farbunterschiede minimiert.

Menschen sind Trichromaten [\[18\]](#), [\[46\]](#), was bedeutet, dass sie drei verschiedene Arten von Augenzapfen besitzen, die jeweils für eine bestimmte Wellenlängenreichweite des sichtbaren Lichts empfindlich sind, dessen Intensität messen und die Information ans Gehirn weiterleiten. [Abb. 18](#) zeigt ein Diagramm der Empfindlichkeit abhängig von der Wellenlänge. Im Gegensatz zu Fischen mit vier [\[19\]](#) und Hunden mit zwei [\[65\]](#) braucht es beim Menschen folglich drei Dimensionen, um jede Farbe verlustfrei repräsentieren zu können.

5.6.1 Farbräume

Dabei gibt es eine Vielzahl an Farbräumen [\[48\]](#), [\[81\]](#), die allesamt Farben als dreidimensionale Vektoren [\[25\]](#) enkodieren. Der womöglich bekannteste [\[6\]](#) ist sRGB, der eine Rot-, eine Grün- und eine Blau-Komponente enthält (und der in [Kap. 4](#) bereits genutzt wurde, um Vektoren zu illustrieren). Daraus setzt sich jede auf einem Computerbildschirm darstellbare Farbe zusammen: Jeder Pixel besteht aus drei Subpixeln, die jeweils Rot, Grün oder Blau in einer bestimmten Intensität anzeigen. Zur Auswahl von Farben eignet sich HSV [\[80\]](#) dagegen besser [\[27\]](#) – hier repräsentieren die Komponenten eines Vektors den Buntton (Hue), die Sättigung der Farbe sowie die Helligkeit (Value). Dieses Farbmodell deckt ebenfalls alle sRGB-Farben ab.

Beide können jedoch nicht dazu genutzt werden, um Farben nach ihrem Aussehen in der realen Welt zu vergleichen – in sRGB sind die Farben

$$\vec{c}_1 := \text{orange} \text{ und } \vec{c}_2 := \text{pink}$$

genauso weit entfernt wie

$$\vec{c}_3 := \text{violet} \text{ und } \vec{c}_4 := \text{green}$$

obwohl die ersten beiden viel ähnlicher erscheinen.

Aus diesem Grund (und weiteren) wurde das perzeptuelle OKLAB-Farbsystem [\[66\]](#) entwickelt. Perzeptuell bedeutet

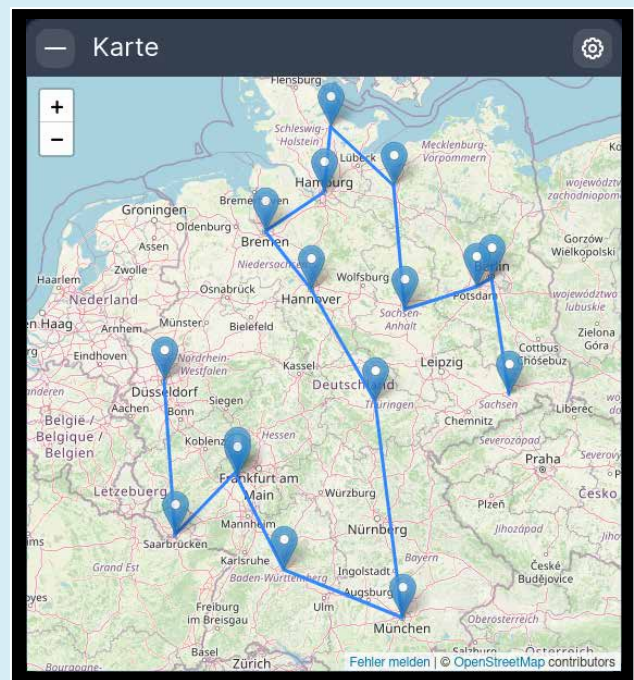


Abb. 17: Kürzester Pfad entlang der 16 Landeshauptstädte Deutschlands als Ketten-sortierung einer Liste ihrer Koordinaten



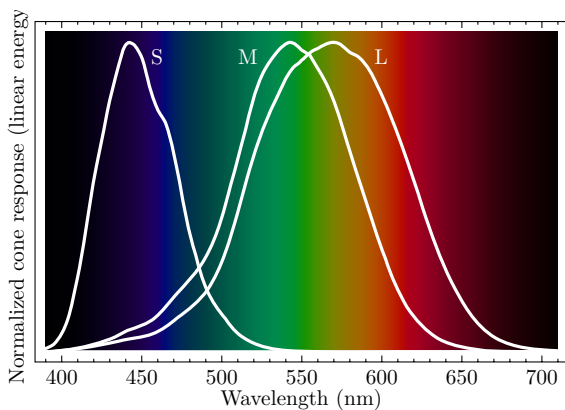


Abb. 18: Empfindlichkeit menschlicher Augenzapfen [11]

zum Zwecke dieser Arbeit, dass die euklidische Distanz zwischen zwei OKLAB-Farbwerten den wahrgenommenen Abstand modelliert, und dass Eigenschaften wie Buntton, Sättigung und Helligkeit experimentellen Daten eher entsprechen [56]. LAB bezieht sich darauf, dass das Farbsystem Farben als Helligkeit (Luminosity) sowie zwei Bunttönen, a und b , repräsentiert. In diesem Farbsystem beträgt $|\vec{c}_1 - \vec{c}_2| \approx 0,39$, während $|\vec{c}_3 - \vec{c}_4| \approx 0,69$.

In der Webanwendung habe ich diese und weitere (linear-sRGB, CMY, HSL, HSV, XYZ und CIELAB) sowie die Konversion zwischen jeden zwei Farbräumen implementiert.

5.6.2 Farbauswahl

Um Farben hinzuzufügen, habe ich einen Farbauswahldialog entworfen (siehe Abb. 19), der intuitiv nutzbar ist und zugleich alle Möglichkeiten der Farbauswahl abdeckt. Jeder Bestandteil ist interaktiv und reaktiv, passt sich also der ausgewählten Farbe direkt bei Veränderung an und ermöglicht durch Auswahl des Farbraums, beliebige Komponenten der Farbe zu verändern. Zudem können verschiedene Listen zur Benennung der Liste ausgewählt werden, wie etwa HTML-, X11- oder RAL-Farben.

5.6.3 Visualisierung

Die Farben werden dem gewählten Farbraum entsprechend in einem dreidimensionalen Koordinatensystem mittels three.js als Kugeln angezeigt, die Kanten des Pfads als zwischen diesen liegende Zylinder. Die Achsen repräsentieren die Bedeutung der Koordinate. Der Projektionstyp ist einstellbar und über die Maus kann die Darstellung skaliert und rotiert werden. In Abb. 20 wird ein 3-opt-optimaler Pfad 18 zufälliger Farben im OKLAB-Farbraum gezeigt; die Abb. 21 zeigt einen Screenshot der vollständigen Anwendung [17] auf 100 Farben in HSL.

6. Fazit und Ausblick

In diesem Projekt ist es mir gelungen, die vergleichsbasierte Sortierung auf Listen n -dimensionaler Daten zu generalisieren und im Anschluss eine Webanwendung zu entwickeln, mithilfe derer diese Sortierung auch in der Praxis anwendbar wird.

Erstaunt war ich zunächst über die Komplexität des Problems. Schließlich werden vergleichsbasierte Sortierungsalgorithmen bereits jetzt universell eingesetzt und die verschiedenen Ansätze sind – auch, wenn immer noch an Mikrooptimierungen gefeilt wird [59] – mittlerweile im algorithmischen Repertoire und der Fachliteratur etabliert [4]. Anders ist das bei der mehrdimensionalen Sortierung, zu der (meiner Kenntnis nach) keine Bibliotheken oder Methoden existieren, die sich explizit zum Ziel setzen, eine Liste mehrdimensionaler Daten zu sortieren. Die NP-Schwere ist nach Erkenntnis der Äquivalenz zum kürzesten Hamilton-Pfad einleuchtend; ohne das Problem graphentheoretisch zu betrachten, hätte ich sie jedoch nicht erkannt.

Besonders fasziniert und motiviert hat mich die Interdisziplinarität der Thematik bei Entwicklung und Ausarbeitung des Projekts. Während ich mit einem rein mathematischen Problem begann, entwickelte es sich über die Graphentheorie hin zu einem praktisch-algorithmischen der Informatik; in der Visualisierung der Ansätze nutze ich das Hooke'sche Gesetz aus der Physik (Kap. 5.4) und die Mercator-Projektion aus der Kartographie (Kap. 5.5). Algorithmen beinhalten neben klassischen Ansätzen der Informatik auch solche mit Bezug zur Materialwissenschaft / Chemie (Simulated Annealing) und in Zukunft womöglich ein Ameisenkolonieverfahren [26] auf Basis biologischer Systeme und Bionik. Nicht zu vernachlässigen sind auch Farbtheorie und -lehre, mit der sich seit der Antike schon Aristoteles [21], [78], da Vinci [1], Newton [20], [77], Werner [90], Goethe [20], [37], [70], Kant [47], [71] und Wittgenstein [74], [89], [91], um nur einige zu nennen, bereits auseinandergesetzt haben.

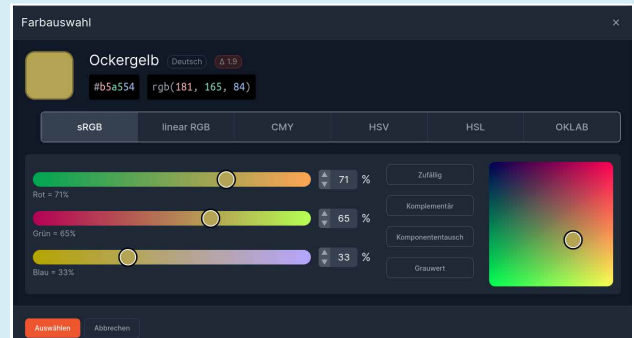


Abb. 19: Der Farbauswahldialog. Aktuell ist Ockergelb im RGB-Farbraum ausgewählt.

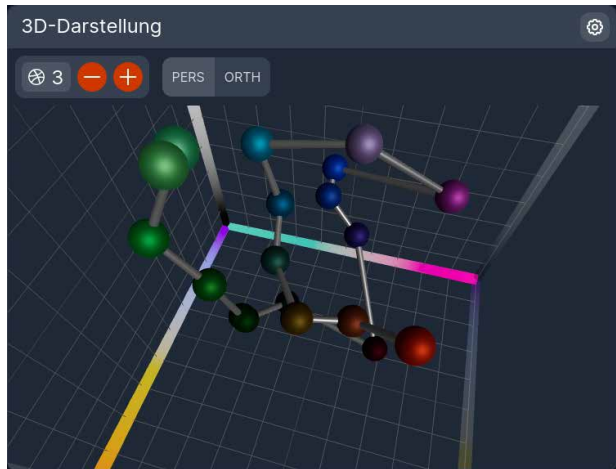


Abb. 20: 3D-Visualisierung

Schon jetzt wird meine inzwischen ausgereifte Anwendung [5] in der Praxis eingesetzt, um verschiedenste Gegenstände nach Farben zu ordnen, Routen zu planen (ich plane beispielsweise, vor meinem Informatikstudium jede der 100 größten Städte Deutschlands zu besuchen, und werde dabei einer von meiner Anwendung errechneten Route folgen) und mithilfe einer Word2Vec [69] -Einbettung [30], [67] Kategorien, Wörter und Begriffe semantisch zu sortieren. In der Zukunft werde ich auf der einen Seite weitere mathematische Eigenschaften der entwickelten Kettensortierung untersuchen und, falls möglich, beweisen – auf der anderen Seite arbeite ich daran, weitere praktische Anwendungsgebiete zu suchen und zu erschließen.

So sprach mich ein Vertreter von Thyssenkrupp an, der bestätigte, dass die mehrdimensionale Sortierung auch in der Vorverarbeitung von Rohstoffen für industrielle Prozesse einsetzbar ist – deshalb arbeite ich inzwischen an einer neuen Unterseite für ebendiese praktisch-industrielle Aufgabe, welche die Vektorsortierungsseite spezialisiert. Zusätzlich konnte ich Kontakt mit einem Vertreter der Textilindustrie aufnehmen, der von der Bedeutung der Farbsortierung im Bereich der Verarbeitung gefärbter Stoffe und des Marketings sprach und ebenfalls die industrielle Relevanz meiner Arbeit hervorhob. Diese Eindrücke motivieren mich, daran weiterzuarbeiten und das Projekt, das Modell sowie die Webanwendung kontinuierlich zu verbessern.

Danksagung

Ich danke meinem langjährigen Projektbetreuer und Mentor in der Informatik-AG, Michael Albrecht, für Motivation, weiterführende Ausarbeitungsideen und die Bereitstellung fachlicher Literatur.

Zudem danke ich meinem Vater, Norman Wojak, für eine sprachliche und gestalterische Überprüfung der Arbeit.

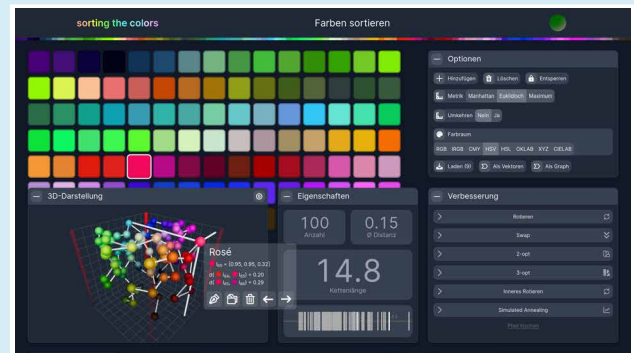


Abb. 21: Ansicht der vollständigen Webanwendung bei Sortierung von 100 Farben im HSV-Farbraum

Schließlich möchte ich auch allen Entwickler:innen der genutzten Open-Source-Bibliotheken danken, die die vorliegende Ausarbeitung in ihrer aktuellen Form maßgeblich erleichtert haben.

Literatur

- [1] Ackerman, J.S. 1980. „On early renaissance color theory and practice“. *Memoirs of the American Academy in Rome* 35:11–44.
- [2] Agafonkin, V., I.S. Ortega, D. Leaver, und andere. „Leaflet – a JavaScript library for interactive maps“. [Online]. Verfügbar unter: <https://leafletjs.com/> [zuletzt geprüft: Januar 6, 2024].
- [3] Akhter, N., M. Idrees, und Furqan-ur-Rehman. 2016. „Sorting Algorithms – A Comparative Study“. *International Journal of Computer Science and Information Security*, 14:930–936.
- [4] Al-Kharabsheh, K.S., I.M. AlTurani, A.M.I. AlTurani, und N.I. Zanoon. 2013. „Review on sorting algorithms a comparative study“. *International Journal of Computer Science and Security (IJCSS)* 7(3):120–126.
- [5] Albrecht, M. 2024. „Nächster Stopp Landesfinale“. [Online]. Verfügbar unter: <https://gymnasium-essen-werden.de/ankuendigungen/n%C3%A4chster-stopp-landesfinale.html> [zuletzt geprüft: März 30, 2024].
- [6] Anderson, M., R. Motta, S. Chandrasekar, und M. Stokes. 1996. „Proposal for a standard default color space for the internet—srgb“. In *Color and imaging conference*. S. 238–245.
- [7] Arora, S. 1998. „Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems“. *Journal of the ACM (JACM)* 45(5):753–782.
- [8] Auger, N., V. Jugé, C. Nicaud, und C. Pivoteau. 2019. „On the Worst-Case Complexity of TimSort“.
- [9] Balaban, I.J. 1995. „An optimal algorithm for finding segments intersections“. In *Proceedings of the eleventh annual symposium on Computational geometry*. S. 211–219.
- [10] Beer, S. 2018. Vergleich und Analyse von Partitionierungsalgorithmen für Quicksort.
- [11] BenRG. 2009. „Cone fundamentals with srgb spectrum“. [Online]. Verfügbar unter: <https://commons.wikimedia.org/wiki/File:Cone-fundamentals-with-srgb-spectrum.svg> [zuletzt geprüft: Januar 6, 2024].

- [12] Bierman, G., M. Abadi, und M. Torgersen. 2014. „Understanding TypeScript“. In European Conference on Object-Oriented Programming. S. 257–281.
- [13] Blume, L. 2021. „Effizienzanalyse des Minimax-Algorithmus im Bezug auf Schach“. [Online]. Verfügbar unter: https://www.jugend-forscht.de/media/2021/project_91120/description/description_2021-02-26_00-15-51.pdf [zuletzt geprüft: Dezember 14, 2023].
- [14] Blume, L. 2022. „Erweiterung klassischer Unterrichtsmedien durch intuitive Webserviceanwendung“. [Online]. Verfügbar unter: https://www.jugend-forscht.de/media/2021/project_93889/description/description_2022-01-16_10-57-20.pdf [zuletzt geprüft: Dezember 16, 2023].
- [15] Blume, L. 2024. „jufo2024-backend: Backend des Jugend forscht-Projekts „sorting-the-colors: Dimensionsbezogene Generalisierung vergleichsbasierter Sortierung““. [Online]. Verfügbar unter: <https://github.com/leo848/jufo2024-backend> [zuletzt geprüft: März 28, 2024].
- [16] Blume, L. 2024. „jufo2024-frontend: Frontend des Jugend forscht-Projekts: „sorting-the-colors: Dimensionsbezogene Generalisierung vergleichsbasierter Sortierung““. [Online]. Verfügbar unter: <https://github.com/leo848/jufo2024-frontend> [zuletzt geprüft: März 28, 2024].
- [17] Blume, L. 2024. „sorting the colors: Farben sortieren“. [Online]. Verfügbar unter: <https://sorting-the-colors.vercel.app/sort-colors> [zuletzt geprüft: März 31, 2024].
- [18] Bompas, A., G. Kendall, und P. Sumner. 2013. „Spotting fruit versus picking fruit as the selective advantage of human colour vision“. *i-Perception* 4(2):84–94.
- [19] Bowmaker, J., und Y. Kunz. 1987. „Ultraviolet receptors, tetrachromatic colour vision and retinal mosaics in the brown trout (*Salmo trutta*): age-dependent changes“. *Vision research* 27(12):2101–2108.
- [20] Burwick, F. 2012. *The damnation of Newton: Goethe's color theory and romantic perception*. Walter de Gruyter.
- [21] Caston, V. 2018. „Aristotle on the Reality of Colors and Other Perceptible Qualities“. *Res Philosophica* 95(1):35–68.
- [22] Chandra, B., H. Karloff, und C. Tovey. 1999. „New results on the old k-opt algorithm for the traveling salesman problem“. *SIAM Journal on Computing* 28(6):1998–2029.
- [23] Chmelka, F., und E. Melan. 1972. „Spannung und Verformung. Das Hooke'sche Gesetz“. *Einführung in die Festigkeitslehre für Studierende des Bauwesens*:26–31.
- [24] Coast, S. 2004. „OpenStreetMap“. [Online]. Verfügbar unter: <https://www.openstreetmap.org/> [zuletzt geprüft: Januar 3, 2024].
- [25] Cohen, J., und T.P. Friden. 1975. „The Euclidean nature of color space“. *Bulletin of the Psychonomic Society* 5(2):159–161.
- [26] Dorigo, M., M. Birattari, und T. Stutzle. 2006. „Ant colony optimization“. *IEEE computational intelligence magazine* 1(4):28–39.
- [27] Douglas, S.A., und A.E. Kirkpatrick. 1999. „Model and representation: the effect of visual feedback on human performance in a color picker interface“. *ACM Transactions on Graphics (TOG)* 18(2):96–127.
- [28] Dörn, S. 2016. „Entwicklung von Computerprogrammen“. *Programmieren für Ingenieure und Naturwissenschaftler: Grundlagen*:95–115.
- [29] Englert, M., H. Röglin, und B. Vöcking. 2014. „Worst case and probabilistic analysis of the 2-Opt algorithm for the TSP“. *Algorithmica* 68(1):190–264.
- [30] Fares, M., A. Kutuzov, S. Oepen, und E. Velldal. 2017. „Word vectors, reuse and replicability: Towards a community repository of large-text resources“. In J. Tiedemann, Hrsg.
- [31] Fateman, R.J. 1989. „Lookup tables, recurrences and complexity“. In *Proceedings of the ACM-SIGSAM 1989 international symposium on Symbolic and algebraic computation*. S. 68–73.
- [32] Fette, I., und A. Melnikov. 2011. „The websocket protocol“.
- [33] Gansner, E.R., E. Koutsofios, S.C. North, und K.-P. Yo. „A method for Drawing Directed Graphs“. [Online]. Verfügbar unter: <https://graphviz.org/documentation/TSE93.pdf> [zuletzt geprüft: Januar 7, 2024].
- [34] Gansner, E.R., und Y. Koren. „Improved Circular Layouts“. [Online]. Verfügbar unter: <https://graphviz.org/documentation/GK06.pdf> [zuletzt geprüft: Januar 7, 2024].
- [35] Gansner, E.R., und S.C. North. 1999. „An open graph visualization system and its applications to software engineering“. *Software – Practice and Experience*. [Online]. Verfügbar unter: <https://graphviz.org/documentation/GN99.pdf> [zuletzt geprüft: Januar 7, 2024].
- [36] Glaubitz, J., D. Rademacher, und T. Sonar. 2019. „Metrik, Norm, Topologie“. In *Lernbuch Analysis 1: Das Wichtigste ausführlich für Bachelor und Lehramt*. Wiesbaden: Springer Fachmedien Wiesbaden, S. 389–411. [Online]. Verfügbar unter: https://doi.org/10.1007/978-3-658-26937-1_13.
- [37] Goethe, J.W. von. 1810. „Zur Farbenlehre“. In *Goethe – Die Schriften zur Naturwissenschaft*. Leopoldina-Ausgabe.
- [38] Gurevich, Y., und S. Shelah. 1987. „Expected computation time for Hamiltonian path problem“. *SIAM Journal on Computing* 16(3):486–502.
- [39] Halbeisen, L., und R. Krapf. 2020. „The Axioms of Set Theory (ZFC)“. In *Gödel's Theorems and Zermelo's Axioms: A Firm Foundation of Mathematics*. Cham: Springer International Publishing, S. 153–171. [Online]. Verfügbar unter: https://doi.org/10.1007/978-3-030-52279-7_13.
- [40] Harris, R., A. Faubert, T.L. Hau, B. McCann, und andere. 2016. „Svelte – cybernetically enhanced web apps“. [Online]. Verfügbar unter: <https://svelte.dev/> [zuletzt geprüft: Januar 3, 2024].
- [41] Harris, R., A. Faubert, T.L. Hau, B. McCann, und andere. „SvelteKit: Web development, streamlined.“. [Online]. Verfügbar unter: <https://kit.svelte.dev/> [zuletzt geprüft: Januar 2, 2024].
- [42] Heijlsberg, A. 2012. „TypeScript: JavaScript with types“. [Online]. Verfügbar unter: <https://www.typescriptlang.org/> [zuletzt geprüft: Januar 2, 2024].
- [43] Held, M., und R. Karp. 1956. „The construction of discrete dynamic programming algorithms“. *IBM Systems Journal* 4(2):136–147.
- [44] Hoare, C.A.R. 1961. „Algorithm 64: Quicksort“. *Communications of the ACM* 4(7):321.
- [45] Hoare, G., und andere. 2015. „A language empowering everyone to build reliable and efficient software.“. [Online]. Verfügbar unter: <https://www.rust-lang.org/> [zuletzt geprüft: Januar 3, 2024].
- [46] Hofer, H., J. Carroll, J. Neitz, M. Neitz, und D.R. Williams. 2005. „Organization of the human trichromatic cone mosaic“. *Journal of Neuroscience* 25(42):9669–9679.
- [47] Jahn, T. 2023. „Zwei mögliche Wege mit dem Dilemma umzugehen“. *Die Eigenarten der Farben*:209–233.
- [48] Joblove, G.H., und D. Greenberg. 1978. „Color spaces for computer graphics“. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*. S. 20–25.
- [49] Johnson, D.S., und L.A. McGeoch. 1997. „The traveling salesman problem: A case study in local optimization“. *Local search in combinatorial optimization* 1(1):215–310.

- [50] Kell, B. „Branch-and-bound algorithm for the traveling salesman problem“. [Online]. Verfügbar unter: <https://www.math.cmu.edu/~bkell/21257-2014f/tsp.pdf> [zuletzt geprüft: März 30, 2024].
- [51] Kingsford, C. „CMSC 451: SAT, Coloring, Hamiltonian Cycle, TSP“. [Online]. Verfügbar unter: <https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/sat.pdf> [zuletzt geprüft: Januar 6, 2024].
- [52] Kivinen, J., M.K. Warmuth, und B. Hassibi. 2006. „The p-norm generalization of the LMS algorithm for adaptive filtering“. *IEEE Transactions on Signal Processing* 54(5):1782–1793.
- [53] Kleinberg, J., und E. Tardos. 2005. *Algorithm Design* 1. Aufl. Pearson Education, Inc.
- [54] Van Laarhoven, P.J., und E.H. Aarts. 1987. *Simulated Annealing*. Springer.
- [55] Lidwell, W., K. Holden, und J. Butler. 2010. „Constraint: A method of limiting the actions that can be performed on a system“. In *Universal principles of design, revised and updated: 125 ways to enhance usability, influence perception, increase appeal, make better design decisions, and teach through design*. Rockport Pub, S. 60–61.
- [56] Lilley, C. 2023. „Color on the Web“. *Fundamentals and Applications of Colour Engineering*:271– 291.
- [57] Läuchli, P. 1991. „Komplexität“. *Algorithmische Graphentheorie*:17–24.
- [58] Mahmoud, H.M. 2000. *Sorting: A distribution theory*. John Wiley & Sons.
- [59] Mankowitz, D.J., A. Michi, A. Zhernov, M. Gelmi, M. Selvi, C. Paduraru, E. Leurent, S. Iqbal, J.-B. Lespiau, A. Ahern, und andere. 2023. „Faster sorting algorithms discovered using deep reinforcement learning“. *Nature* 618(7964):257–263.
- [60] Margot, F. 2009. „Symmetry in integer linear programming“. *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art*:647–686.
- [61] Matsakis, N.D., und F.S. Klock II. 2014. „The Rust language“. In *ACM SIGAda Ada Letters*. S. 103–104.
- [62] McCarthy, L.L., Q. Ye, und D. Shiffman. „home | p5js.org“. [Online]. Verfügbar unter: <https://p5.js.org/> [zuletzt geprüft: Januar 7, 2024].
- [63] McDonnell, C., und andere. „Typescript-first schema validation with static type inference“. [Online]. Verfügbar unter: <https://zod.dev/> [zuletzt geprüft: Januar 4, 2024].
- [64] Morrison, D.R., S.H. Jacobson, J.J. Sauppe, und E.C. Sewell. 2016. „Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning“. *Discrete iv Optimization* 19:79–102. [Online]. Verfügbar unter: <https://www.sciencedirect.com/science/article/pii/S1572528616000062>.
- [65] Neitz, J., T. Geist, und G.H. Jacobs. 1989. „Color vision in the dog“. *Visual neuroscience* 3(2):119–125.
- [66] Ottoson, B. 2020. „A perceptual color space for image processing“. [Online]. Verfügbar unter: <https://bottoson.github.io/posts/oklab/> [zuletzt geprüft: Januar 7, 2024].
- [67] O A. „NLPL word embeddings repository“. [Online]. Verfügbar unter: <http://vectors.nlp.eu/repository/> [zuletzt geprüft: März 30, 2024].
- [68] Peters, O.R.L. 2021. „Pattern-defeating Quicksort“.
- [69] Rehurek, R., und P. Sojka. 2010. „models. word2vec–Word2vec embeddings“. *Gensim*.
- [70] Ribe, N., und F. Steinle. 2002. „Exploratory experimentation: Goethe, Land, and color theory“. *Physics today* 55(7):43–49.
- [71] Riley, C.A. 1995. *Color codes: Modern theories of color in philosophy, painting and architecture, literature, music, and psychology*. UPNE.
- [72] Rollnik, S. 2022. „Vollständige Induktion“. *Übungsbuch fürs erfolgreiche Staatsexamen in der Mathematik: Aufgaben und Lösungen für angehende Lehrkräfte der Sekundarstufe* 1:45–50.
- [73] Scherer, W. 2016. „Anhang C–Landau-Symbole“. *Mathematik der Quanteninformatik: Eine Einführung*:267–268.
- [74] Schwarte, L. 2015. „Farbliche Evidenzerzeugung“. *Pikturale Evidenz*:131–143.
- [75] Seidov, Z. 2005. „A107833“. [Online]. Verfügbar unter: <https://oeis.org/A107833> [zuletzt geprüft: Januar 6, 2024].
- [76] Seiffart, E., und K. Manteuffel. 1974. *Lineare Optimierung*. Springer.
- [77] Shapiro, A.E. 1994. „Artists’ colors and Newton’s colors“. *Isis* 85(4):600–630.
- [78] Silverman, A. 1989. „Color and color-perception in Aristotle’s De Anima“. *Ancient Philosophy* 9(2):271–292.
- [79] Sloane, N. 2024. „The On-Line Encyclopedia Of Integer Sequences“. [Online]. Verfügbar unter: <http://oeis.org/> [zuletzt geprüft: Januar 6, 2024].
- [80] Smith, A.R. 1978. „Color gamut transform pairs“. *ACM Siggraph Computer Graphics* 12(3):12–19.
- [81] Spencer, D.E. 1943. „Adaptation in color space“. *JOSA* 33(1):10–17.
- [82] Stallman, R. 2007. „GNU General Public License“. [Online]. Verfügbar unter: <https://www.gnu.org/licenses/gpl-3.0.en.html> [zuletzt geprüft: Januar 6, 2024].
- [83] Stützle, T. 2003. „The traveling salesman problem: state of the art“. In *TUD-SAP AG Workshop on Vehicle Routing*.
- [84] Tate, J., und M. Atiyah. 2022. „The Millennium Price Problems“. [Online]. Verfügbar unter: <https://www.claymath.org/millennium-problems/> [zuletzt geprüft: Dezember 30, 2023].
- [85] Thielemann, H. 2004. „Klein, aber O“.
- [86] Tolnay, D., und andere. „Serde: Serialization framework for Rust“. [Online]. Verfügbar unter: <https://serde.rs/> [zuletzt geprüft: Januar 4, 2024].
- [87] Waite, W.M., und M. O’Halloran. 1966. „Note on rapid instruction analysis by table lookup“. *The Computer Journal* 9(3):248–249.
- [88] Weitz, E. 2021. „Die Landau-Symbole“. *Konkrete Mathematik (nicht nur) für Informatiker: Mit vielen Grafiken und Algorithmen in Python*:479–494.
- [89] Wenning, W., W. Leinfellner, E. Kraemer, und J. Schänk. 1982. „Wittgensteins ‘Logik der Farbbegriffe’ und die Geometrie des Farbraums“. In *Language and Ontology. Proceedings of the 6th International Wittgenstein Symposium*. Wien.
- [90] Werner, A.G., und P. Syme. 1814. „Werners Nomenklatur der Farben: angepasst an Zoologie, Botanik, Chemie, Mineralogie, Anatomie und die Kunst“.
- [91] Wittgenstein, L., und G.E.M. Anscombe. 1977. *Bemerkungen über die Farben*.
- [92] Zermelo, E. 1908. „Untersuchungen über die Grundlage der Mengenlehre“. In *Mathematische Annalen*. Leipzig: Springer, S. 261–281.
- [93] Zimmermann, T., J. Rüh, B. Wolters, und O. Hohlfeld. 2017. „How HTTP/2 pushes the web: An empirical study of HTTP/2 server push“. In *2017 IFIP Networking Conference (IFIP Networking) and Workshops*. S. 1–9.

Als Schüler*in wissenschaftlich publizieren

Wie auch aus deiner Wettbewerbsarbeit eine zitierfähige Veröffentlichung wird

Was ist eine wissenschaftliche Veröffentlichung?

Wissenschaftliche Publikationen, sogenannte Papers, sind ein zentrales Element wissenschaftlichen Arbeitens. In Papers werden nicht nur Zeitpunkt und Stand einer Erkenntnis öffentlich dokumentiert, sondern auch mit der Wissenschafts-Community geteilt. So lässt man Kolleg*innen derselben Fachrichtung an Ergebnissen teilhaben oder zeigt progressive Forschungsansätze auf.

Was kostet die Veröffentlichung?

Für die Autor*innen fallen keinerlei Veröffentlichungsgebühren (*page charges*) an. Alle Kosten z. B. für Redaktion, Lektorat, Layout, Website und App tragen Verlag und Sponsoren. Verlag ist die Physikalisch-Technische Bundesanstalt PTB, die das Projekt seit Gründung begleitet.

Was ist besonders an einer wissenschaftlichen Veröffentlichung?

Die Besonderheit eines echten, wissenschaftlichen Papers ist, dass es *peer reviewed* ist. Der Begriff setzt sich zusammen aus den englischen Wörtern *peer* für „Kolleg*in“ und *reviewed* für „überprüft“ (*review* = die Überprüfung). Die Arbeit wird also von einem / einer meist anonymen Fachkolleg*in, der oder dem *referee*, auf Schlüssigkeit überprüft. Die Arbeit ist somit gecheckt und kann als Basis für weitere Forschungsvorhaben genutzt werden.

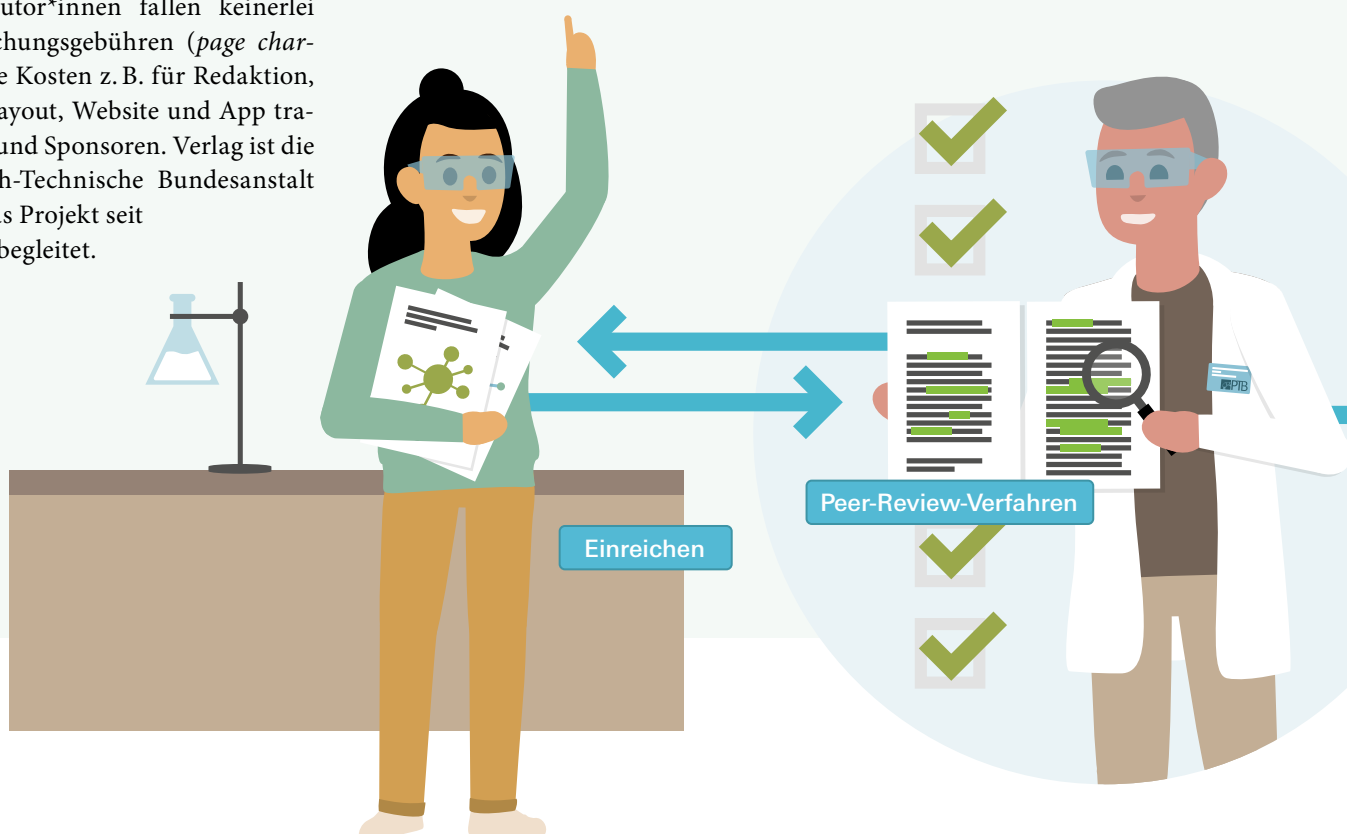


Hast du Fragen? In den FAQs auf der Seite „Für Autor*innen“ findest du Antworten.

www.junge-wissenschaft.ptb.de/fuer-autorinnen

Wieso wissenschaftlich publizieren?

Diese Papers dienen nicht nur dem fachlichen Austausch, sondern auch als Nachweis der erbrachten Leistungen im jeweiligen Spezialgebiet. Wie ein Lebenslauf informiert die Veröffentlichungsliste über den beruflichen Werdegang und wissenschaftlichen Erfolg.



Wie geht das und wie viel Arbeit muss ich investieren?

Die Junge Wissenschaft (JuWi) ist die einzige Plattform, auf der bereits Schüler*innen ein erstes Paper, *peer reviewed*, veröffentlichen können. Das von der JuWi-Chefredaktion eingeleitete und begleitete Peer-Review-Verfahren macht aus deinem Wettbewerbsbeitrag eine zitiertfähige Veröffentlichung. Ein JuWi-Paper ist der Startschuss für deine persönliche Veröffentlichungsliste. Und als erfolgreiche Teilnehmer*in eines Forschungswettbewerbs hast du den Löwenanteil der Arbeit bereits erledigt.

Sende deine Arbeit und die Erstveröffentlichungserklärung an:

Chefredaktion Junge Wissenschaft

Dr.-Ing. Sabine Walter
Paul-Ducros-Straße 7
30952 Ronnenberg

Tel: 05109 / 561508
Mail: sabine.walter@verlag-jungewissenschaft.de



Wie geht es nach dem Einreichen weiter?

Die Chefredakteurin sucht einen geeigneten Fachgutachter*in, der bzw. die, die inhaltliche Richtigkeit der eingereichten Arbeit überprüft und eine Empfehlung ausspricht, ob sie veröffentlicht werden kann (Peer-Review-Verfahren). Das Gutachten wird dir zugeschickt und du erhältst die Möglichkeit, Hinweise des oder der Fachgutachter*in oder eigene Änderungen einzuarbeiten. Die Erfahrung zeigt, dass Arbeiten, die z. B. im Rahmen eines Wettbewerbs wie Jugend forscht die Endrunde erreicht haben, die besten Chancen haben, dieses Peer-Review-Verfahren zu bestehen. Bis hierhin hast du keinerlei Arbeit investiert.

Schließlich kommt die Arbeit in die Redaktion, wird für das Layout vorbereitet und nach der Freigabe als Open-Access-Beitrag, also für jedermann zugänglich, veröffentlicht.

Was bringt es mir?

JuWi-Autor*innen erwerben in der engen Zusammenarbeit mit der Redaktion Kenntnis über den Aufbau einer wissenschaftlichen Arbeit, über wissenschaftlichen Schreibstil, worauf zu achten ist und welche Schritte wann notwendig sind. Autor*innen eines JuWi-Papers haben so sehr früh einen bedeutenden Teil wissenschaftlichen Publizierens erlernt, noch bevor sie an die Hochschule gehen.

Impressum

Junge Wissenschaft

c/o Physikalisch-Technische
Bundesanstalt (PTB)
www.junge-wissenschaft.ptb.de

Redaktion

Dr.-Ing. Sabine Walter,
Chefredaktion Junge Wissenschaft
Paul-Ducros-Str. 7
30952 Ronnenberg
E-Mail: sabine.walter@verlag-jungewissenschaft.de
Tel.: 05109 / 561 508



Sabine Siems, Verlag
E-Mail: sabine.siems@ptb.de
Tel.: 0531 / 592 8202



Design & Satz

Sebastian Baumeister
Art Director / stilsicher.design
E-Mail: baumeister@stilsicher.design
Tel.: 05142 / 302 99 04



Verlag

Dr. Dr. Jens Simon,
Pressesprecher der PTB
Bundesallee 100
38116 Braunschweig
E-Mail: jens.simon@ptb.de
Tel.: 0531 / 592 3006
(Sekretariat der PTB-Pressestelle)

