Optimierung eines Algorithmus zur Generierung von Datensätzen mit gleichen Eigenschaften und unterschiedlicher Darstellung

Algorithm Engineering 2023 Project Paper

Larissa Kolbe

Friedrich Schiller Universität Jena

Deutschland

larissa.kolbe@uni-jena.de

ZUSAMMENFASSUNG

Datensätze, die trotz unterschiedlicher graphischer Darstellung die gleichen statistischen Eigenschaften besitzen, sind eine gute Veranschaulichung der Relevanz graphischer Repräsentationen. Dieses Paper befasst sich mit der Performanceverbesserung des Algorithmus von Matejka und Fitzmaurice [3], welcher eben solcher Datensätze generiert. Während der ursprünglichen Algorithmus dazu ca. zehn Minuten benötigt, generiert meiner in maximal wenigen Sekunden ein Ergebnis. Mein Algorithmus bietet außerdem zusätzlich die Möglichkeit, eigene Zielformen hochzuladen, um so beliebige Darstellungen zu erzeugen. Zur Verbesserung der Performance habe ich zunächst die Hotspots des ursprünglichen Algorithmus ausfindig gemacht und anschließend verschiedene Ansätze zur Optimierung verglichen. Dabei hat sich unter anderem gezeigt, dass der naive Ansatz in vielen Fällen bereits performanter ist als komplexere Lösungen.

KEYWORDS

Performanceverbesserung, Scatter Plots, Anscombe, Visualisierung

1 EINLEITUNG

1.1 Hintergrund

Bei der Arbeit mit Datensätzen spielt die graphische Repräsentation eine wichtige Rolle. Klare Formen und Strukturen besitzen eine deutlich größere Aussagekraft, als unstrukturierte Punktewolken. Anscomes Quartett [1] aus dem Jahr 1973 ist eine der ältesten und meistreferenzierten Arbeiten in diesem Zusammenhang. Es besteht aus vier gleichgroßen Datensätzen, die alle über dieselben statistischen Eigenschaften (Durchschnitt, Standardabweichung und Korrelation) verfügen. Dennoch besitzen sie alle von Grund auf verschiedene graphische Darstellungen, wie in Abbildung 1-A abgebildet. Während Datensatz I wie eine grob lineare Punktewolke wirkt, ist Datensatz II wie eine Parabel geformt. Die Punkte in Datensatz III und IV bilden bis auf je eines Ausreißers eine klare Linie, die jedoch bei dem einen Datensatz linear und bei dem anderen vertikal verläuft. Im Vergleich sieht man darunter Datensätze, die ebenfalls dieselben statistischen Eigenschaften wie Anscomes Quartett besitzen, deren Darstellung allerdings keinem bestimmten Muster folgt. Dadurch fehlt es ihnen im Gegensatz zu Anscomes Quartett an Aussagekraft. [3]

Matejka und Fitzmaurice [3] haben angelehnt an Anscomes Quartett einen Algorithmus entworfen, der die statistischen Eigenschaften eines übergebenen Datensatzes berechnet und ihn unter Beibehaltung dieser umwandelt. Dazu stellen sie ein Set an möglichen Zielformen bereit, die der Datensatz annehmen kann. Die Python-Implementierung dieses Algorithmus ist jedoch sehr langsam, sodass die Umwandlung eines Datensatzes auf einem herkömmlichen Laptop ca. 10 min benötigt. Außerdem bietete die Methode durch die wenigen, hartkodierten Zielformen nur wenig Freiraum. Mein Algorithmus orientiert sich an Matejka und Fitzmaurices Methode, erzielt allerdings unter anderem durch die Implementierung in C++ eine deutlich bessere Performance. Durch die Möglichkeit, selbst eine Zielform hochzuladen, bietet er außerdem mehr Freiraum beim Erstellen der Datensätze.

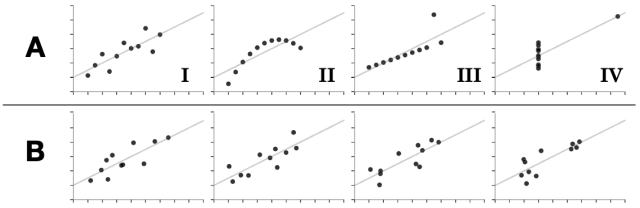


Abbildung 1: A zeigt Anscomes Quartett, B vier unstrukurierte Datensätze. Alle acht Datensätze haben die gleichen statistischen Eigenschaften (Durchschnitt, Standardabweichung, Korrelation).

(Abbildung aus [3] entnommen)

1.2 Verwandte Arbeiten

Auch 50 Jahre nach ihrer Veröffentlichung sind die Datensätze aus Anscomes Quartett [1] nach wie vor aussagekräftig und werden oft als Veranschaulichung der Wichtigkeit visueller Methoden genutzt [3]. Trotz ihrer Bekanntheit ist jedoch nichts über die Methode zur Erstellung der Datensätze bekannt.

Chatterjee und Firat [4] haben 2007 erstmals eine Methode zur Erstellung solcher Datensätze veröffentlicht. Dazu wurden zunächst 1.000 zufällige Datensätze mit gleichen statistischen Eigenschaften erstellt und kombiniert. Anschließend wurden sie mithilfe einer Zielfunktion so mutiert, dass die "graphische Unähnlichkeit" zwischen der ursprünglichen und der resultierenden graphischen

Abbildung 2: Einige Beispielbilder, die mit meinem Algorithmus aus dem linken Inputdatensatz erstellt wurden. Sie alle haben bis auf zwei Nachkommastellen Genauigkeit die gleichen statistischen Eigenschaften ($mean_x = 51,42; mean_y = 50,04; stdDev_x = 28,76; stdDev_y = 29,75$).

Darstellung maximal wird. Dadurch sahen die Ergebnisse dem ursprünglichen Datensatz zwar nicht ähnlich, hatten allerdings auch keine feste Struktur, wodurch es ihnen im Vergleich zu Anscomes Quartett an Aussagekraft mangelt. Mein Algorithmus hebt sich von diesem dadurch ab, dass er die Möglichkeit bietet, eigene Zielformen hochzuladen oder aus einer Reihe vorgegebener Formen zu wählen. Er ist außerdem flexibel und kann bei Bedarf um beliebige statistischen Eigenschaften erweitert werden.

2008 stellten Govindaraju und Haslett eine Methode vor, um Datensätze auf die Durchschnittswerte ihrer Stichproben zurückzuführen [6]. Dabei wurde die lineare Regressionsformel beibehalten. Ein Jahr später erweiterten sie ihr Verfahren um "geklonte" Datensätze zu erstellen [5]. Dabei blieben zusätzlich zur linearen Regression auch die Durchschnittswerte des Datensatzes unverändert. Die Methode war so konstruiert, dass die erstellten Datensätze dem ursprünglichen ähnlich sahen. Abhängig von der gewählten Zielform kann mein Algorithmus ebenfalls Datensätze mit ähnlichen graphischen Eigenschaften erzeugen. Der primäre Fokus liegt bei mir jedoch auf der möglichen Vielseitigkeit der Ergebnisgraphen.

Zu guter Letzt haben Matejka und Fitzmaurice 2017 [3] einen Algorithmus entwickelt, der die Punkte eines Eingabedatensatzes so verschiebt, dass sie eine bestimmte Zielform annehmen ohne dabei ihre statistischen Eigenschaften zu verändern. Dazu werden die Punkte innerhalb mehrere Iterationen stückweise der Zielform angenähert. Nach jeder Bewegung wird geprüft, ob der Punkt der Zielform näher kam und, ob die statistischen Eigenschaften unverändert bleiben. Nur dann wird der neue Wert übernommen. Die statistischen Eigenschaften können dabei beliebig gewählt werden. Der Grundablauf meines Algorithmus stimmt mit diesem überein. Allerdings habe ich ihn an einigen Stellen leicht abgewandelt, um die Performance zu verbessern. Weiterhin bietet mein Algorithmus im Gegensatz zu dem von Matejka und Fitzmaurice die Möglichkeit, eine eigene Zielform hochzuladen, statt nur aus hartkodierten Möglichkeiten auszuwählen.

1.3 Überblick

In Kapitel 2 werde mich zunächst dem von mir angepassten Algorithmus widmen. Dabei werde ich insbesondere auf das Einlesen der Daten sowie den Aufbau des Algorithmus selbst und dessen einzelner Funktion eingehen. Anschließend werde ich in Kapitel 3 einige weitere Ansätze zur Verbesserung der Performance testen und vergleichen.

2 DER ALGORITHMUS

Wie bereits in den vorherigen Abschnitten erwähnt, habe ich mich weitestgehend an dem von Matejka und Fitzmaurice [3] vorgestellten Algorithmus orientiert. Zunächst werden ein Inputdatensatz und eine Zielform eingelesen. Anschließend werden die Punkte des Datensatzes iterativ in Richtung der Zielform bewegt, wobei die ursprünglichen statistischen Eigenschaften stets beibehalten werden.

Im Gegensatz zu Matejka und Fitzmaurice habe ich den Algorithmus in C++ statt in Python implementiert, um eine bessere Performance zu erreichen. Im Folgenden werde ich hauptsächlich auf die Änderungen eingehen, die ich im Vergleich zu Matejka und Fitzmaurice vorgenommen habe.

2.1 Einlesen der Daten

Bevor der eigentliche Algorithmus beginnt, werden über ein User Interface die zu verwendenden Daten eingelesen. Der User hat die Möglichkeit, den Inputdatensatz und die Zielform entweder selbst hochzuladen oder sie aus einer Auswahl vorgegebener Möglichkeiten in verschiedenen Bildgrößen zu wählen. Um das Auslesen der Daten zu vereinfachen, können nur .ppm-Bilder hochgeladen werden.

Wurden die zu verwendenden Bilder gewählt, werden die Daten aus ihnen ausgelesen. Dazu wird über alle Pixel iteriert und ihre Farbwerte extrahiert. Ist der Pixel schwarz oder nahezu schwarz, wird seine Position in Form von x- und y-Koordinaten in einem Vektor gespeichert. Dazu verwende ich ein Objekt mit Float-Attributen für die Koordinaten. Das Auslesen der Daten erfolgt sowohl für den Inputdatensatz als auch für die Zielform gleicherweise. Die dabei entstandenen Koordinatenvektoren `initialDS` und `targetShape` werden anschließend dem Algorithmus übergeben und modifiziert beziehungsweise zum Vergleich verwendet. Um den Cache optimal auszunutzen, verwende ich dafür keine normalen Vektoren oder Arrays, sondern auf 64 Byte alignte Vektoren.

Nachdem die Daten eingelesen wurden, hat der User die Möglichkeit, die Konfigurationen auf denen der Algorithmus laufen soll einzusehen und zu verändern. Dadurch ist es ermöglicht, die Daten mit beliebiger Genauigkeit zu verändern oder die Anzahl an Iterationen zu beeinflussen. Da die optimalen Einstellungen abhängig von den Daten sind, müssen diese Werte eventuell angepasst werden, um ein optimales Ergebnis zu erzielen. In den Tests erwiesen sich die aktuellen Standardwerte als effektiv. Um an dieser Stelle nicht zu weit auszuschweifen, werde ich die einzelnen Konfigurationsmöglichkeiten später an gegebener Steller näher erläutern.

2.2 Verarbeitung

Algorithm 1 Pseudocode des Algorithmus

```
1:  initialProperties ← calcStats(initialDS)
2:  currentDS ← initialDS
3:  for n iterations do parallel
4:      indexToMove ← random()
5:      movedPoint ← perturb(currentDS[indexToMove],
            targetShape, temp)
6:    critical section:
7:      testDS ← currentDS
8:      testDS[indexToMove] ← movedPoint
9:      if isErrorOk(testDS, initialProperties) then
10:         currentDS[indexToMove] ← movedPoint
11:     end if
12:   end critical section
13: end for
14:
15: procedure perturb(pointToMove, targetShape, temp)
16:     while True do
17:         modifiedPoint ← movePoint(pointToMove)
18:         if isBetterFit(modifiedPoint, pointToMove, targetShape)
              or random() < temp then
19:             return modifiedPoint
20:         end if
21:     end while
22: end procedure
```

Nachdem die Daten eingelesen wurden, folgt ihre Modifizierung zur der Zielform hin. Der Ablauf dieses Prozesses ist in Algorithmus 1 dargestellt. `initialDS` ist der wie in Abschnitt 2.1 beschriebene Vektor aus Koordinaten des Inputdatensatzes und `targetShape` analog der der Zielform. Anhand von `initialDS` werden zu Beginn in calcStats die zu erhaltenden statistischen Eigenschaften berechnet. Standardmäßig sind das bei mir der Durchschnitt und die Standardabweichung der x- und y-Koordinaten. Der Algorithmus ist allerdings beliebig um weitere Eigenschaften erweiterbar. Die ermittelten Werte werden anschließend auf die konfigurierte Anzahl an Nachkommastellen gerundet, was bei mir standardmäßig zwei sind. Diese Werte werden später in isErrorOk als Richtlinie verwendet.

Im nächsten Schritt wird der Datensatz in einer parallelen Schleife modifiziert. Die Anzahl an Iterationen lässt sich beliebig konfigurieren und liegt standardmäßig bei 200.000. Für die meisten Datensätze lässt sich damit ein gutes Ergebnis erzielen. In jeder Iteration wird zunächst ein zufälliger Punkt bestimmt. Dieser wird der Funktion perturb übergeben und dort mit movePoint um eine zufällige Entfernung bewegt. Die maximale Bewegung pro Iteration kann in den Konfigurationen bestimmt werden und liegt standardmäßig bei 1. Anders als Matejka und Fitzmaurice übergebe ich nur den zu modifizierenden Punkt, statt des ganzen Datensatzes, um die zu übergebende Datenmenge zu minimieren. Dadurch kann bei meinem Algorithmus pro Iteration nur ein Punkt modifiziert werden.

Sobald die neuen Koordinaten des Punktes bestimmt sind, wird in isBetterFit geprüft, ob der Punkt der Zielform nun näher ist als zuvor. Dazu wird die minimale euklidische Distanz zwischen den modifizierten bzw. den ursprünglichen Koordinaten und den Punkten der Zielform ermittelt und verglichen. Ist die Distanz kleiner geworden, werden die modifizierten Koordinaten zurückgegeben. Andernfalls wird die Bewegung zurückgesetzt und der Punkt erneut bewegt. Im Gegensatz zu Matejka und Fitzmaurice berechne ich die *Fitness* der beiden Punkte nicht getrennt, sondern in einem Durchgang. So muss der Zielformvektor nur einmal durchgegangen werden, statt doppelt. Des Weiteren wird in meinem Algorithmus nur die minimale Distanz des übergebenen Punktes ermittelt, während Matejka und Fitzmaurice die durchschnittliche minimale Distanz aller Punkte des Datensatzes zur Zielform berechnen.

Um zu vermeiden, dass der Algorithmus in einer lokalen optimalen Lösung stecken bleibt, nutzen Matejka und Fitzmaurice ein Simulated-Annealing-Verfahren (vgl. [2]). Eine Bewegung kann dadurch auch dann akzeptiert werden, wenn sie sich von der Zielform entfernt. Dazu wird eine *Temperatur* (`temp`) eingeführt, die mit jeder Iteration kleiner wird. Ist der Wert einer Zufallszahl zwischen 0 und 1 kleiner als diese *Temperatur*, so wird die Bewegung trotz verschlechterter *Fitness* akzeptiert. In meinem Algorithmus können der Anfangs- und Endwert der *Temperatur* beliebig konfiguriert werden. Standardmäßig liegen sie wie bei Matejka und Fitzmaurice bei 0,4 bzw. 0,01.

Zu guter Letzt werden in isErrorOk die statistischen Eigenschaften des modifizierten Datensatzes ermittelt und mit den initialen Werten verglichen. Dazu wird eine Kopie des Datensatzes erstellt, die den modifizierten Wert enthält. Erst, wenn die Modifizierung akzeptiert wurde, wird der Wert in `currentDS` gespeichert. Das ist der Fall, wenn die absolute Differenz zwischen den einzelnen Eigenschaften kleiner gleich der konfigurierten erlaubten Abweichung (standardmäßig 0) ist.

Da der verwendete Datensatz gleichzeitig auch von den anderen Threads gelesen und bearbeitet wird, kann es hier zu einer *Race Condition* mit *Lost Updates* kommen. Um das zu vermeiden, wird der Abschnitt vom Kopieren des Datensatzes bis zum Überschreiben des Wertes in `currentDS` als *Critical Section* markiert. Dadurch wird zwar die Korrektheit sichergestellt, es verlangsamt jedoch auch die Laufzeit durch die Berechnung der statistischen Eigenschaften in isErrorOk.

Eine etwas schnellere Alternative wäre, die *Critical Section* nur um die Zeile zu hüllen, in der der Datensatz überschrieben wird (Zeile 10). Dann muss jedoch, statt nur dem veränderten Punkt, der gesamte Datensatz `currentDS` mit `testDS` überschrieben werden. Andernfalls kommt es zu fehlerhaften Updates, die dazu führen, dass isErrorOk in so gut wie allen Fällen `false` zurückgibt.

3 EXPERIMENTE

Um die Performance des Algorithmus weiter zu verbessern, habe ich zunächst die Hotspots ausfindig gemacht und verschiedene Implementierungen der entsprechenden Funktionen getestet und verglichen.

Alle Messungen wurden mit den Standardkonfigurationen und möglichst großen Datensätzen durchgeführt. Der Inputdatensatz bestand bei allen Messungen aus 1.500 Punkten, während die Zielform 10.275 Datenpunkte besaß. Weiterhin habe ich die Laufzeiten sowohl mit und als auch ohne Optimierungsflags gemessen. Dies ist in den entsprechenden Tabellen je in der Spalte *Flags* angegeben. Die verwendeten Optimierungsflags sind: `'-march=native -O3 -Ofast -ffast-math'`

Die reinen Laufzeitmessungen wurden auf einem MacBook Pro 2017 mit 2,8 GHz Quad-Core Intel Core i7 Prozessor und 16 GB Arbeitsspeicher durchgeführt. Die Profilermessungen wurden auf einem Linux-Betriebssystem mit Intel Core i5-6500 Prozessor und 12 GB RAM gemacht.

3.1 Gesamter Algorithmus

Um Variablenzugriffe und -übergaben möglichst effizient zu gestalten, habe ich diese wo möglich als Konstanten definiert und Vektoren als Pointer mit dem `__restrict__` Keyword übergeben, statt als Kopie. Bei einer Messung mit dem clang-Compiler verbesserte sich die Gesamtlaufzeit dadurch von ca. 3 sek auf nur 1,8 sek. Pro Iteration brauchte der Algorithmus dadurch nur noch ca. 71 µs statt knapp 120 µs.

Der Algorithmus ohne Parallelisierung hat, wie in Tabelle 1 zu sehen eine relativ schlechte Laufzeit und nur sehr geringe effektive CPU-Auslastung. Um diese zu verbessern, habe ich die Iterationsschleife (vgl. Zeile 3 in Algorithmus 1) in ein `'omp parallel for'`-Pragma von OpenMP gehüllt und verschiedener *Schedule*-Varianten verglichen, die statische und die dynamische Parallelisierung. Für Letztere habe ich verschiedene Chunkgrößen getestet, wobei sich eine Größe von acht als am effektivsten erwiesen hat.

Wie die Tabelle zeigt, sind die parallelen Varianten mehr als 2,6 bzw. viermal so schnell, wie die nicht-parallele. Ähnliche steht es auch um die effektive CPU-Auslastung. In beiden Fällen erzielt die dynamische Parallelisierung die mit Abstand besten Ergebnisse.

Weniger gut schneidet der Algorithmus in den Punkten *Retiring* und *Backend Bound* ab, wobei sich hier ein deutlicher Unterschied zwischen den verschiedenen Datensatzgrößen abbildet. Während die Werte bei großer Datenmenge für alle Ansätze bei ca. 50% liegen, unterscheiden sie sich bei kleiner Datenmenge deutlich voneinander. Für diese liegen sowohl die *Retiring*-Rate als auch die Backend-Gebundenheit bei den parallelen Ansätzen in einem guten Bereich. Die Ressourcen und Kerne werden hier also effizient genutzt. Das hängt vermutlich damit zusammen, dass die kleineren Datensätze den Cache effizienter ausnutzen können.

Tabelle 1: Profiling-Ergebnisse. Die Spalte *Parallelisierung* zeigt an, ob der Algorithmus sequentiell oder parallel mit statischem bzw. dynamischem Scheduling lief. Fette Werte implizieren Optimierungsbedarf an den entsprechenden Stellen. Der obere Abschnitt wurde mit den in Abschnitt 3 beschriebenen Datensätzen gemessen, der untere auf kleineren Datensätzen (Inputdatensatz 500 Punkten, Zielform 1.321 Datenpunkten).

Parallelisierung	Laufzeit in sek	Retiring	Backend Bound	Core Bound	CPU-Auslastung
keine	8,706	**48,5%**	**50,1%**	**48,4%**	**23,6%**
static	3,321	**50,4%**	**48,6%**	**46,1%**	**60,1%**
dynamic	2,081	**55,6%**	**43,2%**	**41,0%**	81,3%
keine	2,008	**53,9%**	**38,0%**	**34,2%**	**20,8%**
static	0,600	80,2%	13,5%	12,4%	**58,7%**
dynamic	0,505	88,9%	**3,4%**	2,9%	77,5%

Weiterhin ist auffällig, dass die CPU-Auslastung beim dynamischen *Scheduling* deutlich effizienter ist, als beim statischen. Das hängt damit zusammen, dass die Laufzeit der einzelnen Iterationen nicht gleichmäßig verteilt, sondern zufällig ist. Sie hängt davon ab, wie oft der zu bewegende Punkt erneut verschoben werden und seine aktuelle *Fitness* bestimmt werden muss.

Die Laufzeit der einzelnen Funktionen des Algorithmus ist in Tabelle 2 abgebildet. Für die Messungen wurde dynamische Parallelisierung verwendet. Man kann sehen, dass die Anzahl an *Runs* ca. 2,4-mal so hoch ist, wie die an Iterationen. Das heißt, durchschnittlich wird ein Punkt während einer Iteration 2,4-mal bewegt und folglich auch isBetterFit entsprechend oft aufgerufen. Da diese Funktion zusätzlich eine relativ hohe Laufzeit hat, stellt sie ein Bottleneck des Algorithmus dar. Näheres dazu in Kapitel 3.2.

Auch die Funktion zur Berechnung der statistischen Eigenschaften hat eine relativ hohe Laufzeit, da der Inputdatensatz zur Berechnung der Durchschnitts- und der Standardabweichungswerte zweimal durchlaufen wird. In Kapitel 3.3 werde ich näher darauf eingehen.

Tabelle 2: Durchschnittliche Gesamtlaufzeit des Algorithmus in Sekunden und seiner Komponenten in Mikrosekunden (µs). Die Zeiten beziehen sich je auf eine Ausführung der entsprechenden Funktion. *Comp.* zeigt den verwendeten Compiler an. *Runs* gibt an, wie oft die Funktionen movePoint und isBetterFit aufgerufen wurden.

Comp.	Flags	Ges.	Iteration	calcStats	movePoint	isBetterFit	Runs
gcc	ohne	5,64	224,64	5,61	0,175	88,55	484.549
gcc	mit	1,97	77,26	2,14	0,136	29,69	478.368
clang	ohne	1,81	71,37	3,89	0,202	25,75	469.173
clang	mit	1,62	62,48	3,72	0,196	23,32	467.062

3.2 Berechnung der *Fitness*

Die *Fitness* einer Bewegung wird in isBetterFit berechnet und verglichen. Dazu wird die minimale euklidische Distanz zwischen dem Punkt vor und nach der Bewegung und den Elementen des Zielformvektors ermittelt. Dies wird in jeder Iteration mindestens einmal gemacht. Da die Funktion den gesamten Zielformvektor durchgeht, kann sie viel Zeit in Anspruch nehmen und bildet damit ein Bottleneck des Algorithmus. Ich habe daher verschiedene Ansätze zur Implementierung dieser Funktion getestet und verglichen.

Der naive Ansatz durchläuft den Zielformvektor in einer einfachen for-Schleife, wie in Algorithmus 2 beschrieben. Mein erster Ansatz war, diese for-Schleife mit OpenMP zu parallelisieren (*'Parallel'*). Dazu habe ich das `'omp parallel for'`-Pragma mit einer `min`-Reduction verwendet.

Algorithm 2 : Naive Implementierung zum Vergleichen der *fitness* zweier Punkte

```
1: procedure isBetterFit(p1, p2, targetShape)
2:     minDistP1 ← getDistance(p1, targetShape[0])
3:     minDistP2 ← getDistance(p2, targetShape[0])
4:     for point in targetShape do
5:         minDistP1 ← min(minDistP1, getDistance(p1, point)
6:         minDistP2 ← min(minDistP2, getDistance(p2, point)
7:     end for
8:     return minDistP1 < minDistP2
9: end procedure
```

Ein weiterer Ansatz war die Nutzung von *Vector Intrinsics*. Ich habe sowohl 128-Bit (*'VI128'*) als auch mit 256-Bit Vektoren (*'VI256'*) getestet. Erstere können je die Koordinaten von zwei Punkten speichern und damit die Entfernung beider übergebener Punkte gleichzeitig berechnen. Letztere hingegen können die von vier Punkten gleichzeitig berechnen und benötigen damit nur halb so viele Iterationen. Testweise habe ich diese Ansätze zusätzlich mit *Loop Unrolling* um dem Faktor 8 (*'VIxxx-Lu'*), Parallelisierung wie oben beschrieben (*'VIxxx-Par'*) sowie einer Kombination aus beidem (*'VIxxx-LuPar'*) verbunden.

Die mit diesen Ansätzen je ermittelte Performance ist in Tabelle 3 dargestellt. Es lässt sich erkennen, dass der naive Ansatz mit Abstand am schnellsten ist, gefolgt von den einfachen *Vector-Intrinsics*-Implementierungen. Die Varianten mit *Loop Unrolling* und Parallelisierung hingegen sind in allen Fällen deutlich langsamer, als die einfachen Varianten. Das hängt unter anderem damit zusammen, dass mit Ausnahme von *'VI256-LuPar'* bei diesen Ansätzen die CPU-Auslastung deutlich ineffektiver ist.

Eine interessante Erkenntnis ist, dass die Performance der 256-Bit-Vektoren in den meisten Fällen deutlich schlechter ist, als die der 128-Bit-Vektoren. Die einzige Ausnahme stellt auch hier die Version dar, die sowohl *Loop Unrolling* als auch Parallelisierung nutzt. Während die Vektor-*Capacity*-Nutzung bei den 256-Bit-Vektoren je deutlich besser ist, haben die 128-Bit-Vektoren eine bessere *Retiring*-Rate und sind weniger Backend-gebunden.

Insgesamt scheint jedoch die naive Implementierung am performantesten zu sein. Dies lässt sich vermutlich damit erklären, dass sie durch ihre Einfachheit bereits ausreichend vom Compiler optimiert werden kann. Tests mit anderen Compilern haben selbst ohne Optimierungsflags dasselbe ergeben. Einzig beim gcc-Compiler konnte die Funktion ohne entsprechende Flags nicht optimiert werden, wodurch die *Vector-Intrinsics*-Implementierungen hier schneller waren.

3.3 Berechnen der stat. Eigenschaften

Die Performance der Berechnung der statistischen Eigenschaften hängt stark von ihrer Wahl und Anzahl ab. In meiner Implementierung werden zunächst die Durchschnittswerte der x- und y-Koordinaten berechnet. Mit diesen wird anschließend in einer weiteren for-Schleife die Standardabweichung ermittelt. Insgesamt wird der Datensatz also zweimal komplett durchlaufen.

Ich habe auch einen Ansatz getestet, in dem der Datensatz nur einmal durchlaufen wird. Das hat allerdings dazu geführt, das die berechneten Eigenschaften nur in den wenigsten Fällen übereinstimmten, wodurch kein zufriedenstellendes Ergebnis erzielt werden konnte. Daher habe ich diesen Ansatz wieder verworfen.

Tabelle 3: *Profiling*-Ergebnisse auf verschiedenen Implementierungen für isBetterFit. Fette Werte implizieren Optimierungsbedarf an den entsprechenden Stellen. Messungen wurden auf dem Intel-Compiler mit Optimierungsflags durchgeführt.

Ansatz	Laufzeit in sek	Retiring	Backend Bound	Core Bound	Vektor Capacity Nutzung	effektive CPU-Auslastung
Naiv	2,140	45,8%	**51,7%**	**48,9%**	100%	93,8%
VI128	2,763	69,2%	**29,1%**	**26,9%**	**29,6%**	89,2%
VI256	3,317	**60,0%**	**31,3%**	**27,5%**	**63,5%**	94,6%
Parallel	3,943	**45,6%**	**51,9%**	**48,1%**	100%	**52,1%**
VI128-Lu	4,401	55,5%	**43,1%**	**40,9%**	**46,9%**	**68,5%**
VI128-Par	4,531	68,3%	**28,9%**	**26,1%**	**29,6%**	**61,7%**
VI256-Par	5,840	48,5%	**40,0%**	**34,8%**	**63,2%**	**59,0%**
VI256-LuPar	6,189	**64,2%**	**34,2%**	**29,0%**	**43,2%**	93,1%
VI256-Lu	7,140	**42,3%**	**55,5%**	**48,1%**	**82,7%**	**44,6%**
VI128-LuPar	8,198	**68,2%**	**30,3%**	**27,9%**	**36,1%**	**68,7%**

Als nächstes habe ich den Einfluss von OpenMP Pragmas auf die Performance getestet. Mein erster Ansatz war, die for-Schleifen je in ein `'omp parallel for'`-Pragma mit Reduktion auf den Durchschitts- bzw. Standardabweichungsvariablen zu hüllen. Der zweite war die Nutzung des `'SIMD'`-Pragmas. Hierbei habe ich ebenfalls eine Reduktionsklausel verwendet und zusätzlich angegeben, dass der verwendete Vektor *aligned* ist.

Tabelle 4 zeigt die Laufzeit dieser Ansätze im Vergleich zur naiven Implementierung. Auch hier ist der parallele Ansatz in fast allen Fällen deutlich langsamer. Weiterhin kann man erkennen, dass ohne Optimierungsflags der SIMD-Ansatz schneller ist, während andernfalls der naive Ansatz performanter ist. Während die Performance der SIMD-Implementierung jedoch in allen Fällen weitestgehend gleich ist, gibt es beim naiven Ansatz abhängig vom Compiler und den Optimierungsflags große Schwankungen. Daher würde ich an dieser Stelle die Verwendung des `SIMD`-Pragmas empfehlen.

Tabelle 4: Durchschnittlichen Laufzeiten verschiedener Implementierungen zur Berechnung der stat. Eigenschaften des Datensatzes in Mikrosekunden (µs). *Comp.* zeigt den verwendeten Compiler an.

Comp.	Flags	Naiv	SIMD	Parallel
gcc	ohne	5,53	3,10	7,41
gcc	mit	2,16	3,33	3,43
clang	ohne	3,99	3,71	6,68
clang	mit	3,67	3,73	5,80

4 ZUSAMMENFASSUNG UND AUSBLICK

Mein Ziel war das Verbessern der Performance des von Matejka und Fitzmaurice [3] vorgestellten Algorithmus zum Verändern der visuellen Darstellung eines Datensatzes in Richtung einer bestimmten

Zielform bei gleichbleibenden statistischen Eigenschaften. Mein Algorithmus benötigt zur Verarbeitung eines Inputdatensatz mit 1.500 Datenpunkten und eines Zielformvektors mit über 10.000 Elementen weniger als zwei Sekunden. Für kleinere Datensätze benötigt er nicht einmal 0,5 Sekunden. Matejka und Fitzmaurices Algorithmus hingegen benötigt bei gleich vielen Iterationen und deutlich kleineren Datensätzen ca. zehn Minuten. Mein Algorithmus hat damit eine deutlich bessere Performance.

Trotz der stark verbesserten Performance gibt es, wie die Profiler-Ergebnisse zeigen, an einigen Stellen noch Optimierungsmöglichkeiten. In weiteren Arbeiten könnte man sich mit diesen Punkten beschäftigen.

Da jeder Datenpunkt in meiner Implementierung einem Pixel in der graphischen Darstellung entspricht, können nur ganzzahlige Werte dargestellt werden. Während des Modifizierungsprozesses arbeite ich jedoch mit Float-Werten, da die Bewegungen andernfalls einen zu großen Einfluss auf die statistischen Eigenschaften hätten. Aus diesem Grund müssen die Werte vor dem Export des Datensatzes auf ganzzahlige Werte gerundet werden. Das kann jedoch die statistischen Eigenschaften und damit die Korrektheit des Ergebnisses beeinflussen. Um diese sicherzustellen, müsste man daher die Darstellung der Daten anpassen.

Um optimale Ergebnisse zu erzielen, sollten die hochgeladenen Bilder des Inputdatensatzes und der Zielform möglichst gleich groß sein. Um mehr Freiraum in der Bildgenerierung zu bieten, könnte man stattdessen versuchen die Zielform durch Skalierung an die Größe des Inputbildes anzupassen.

LITERATUR

[1] F. J. Anscombe. 1973. Graphs in Statistical Analysis. *The American Statistician* 27, 1 (1973), 17–21. https://doi.org/10.2307/2682899

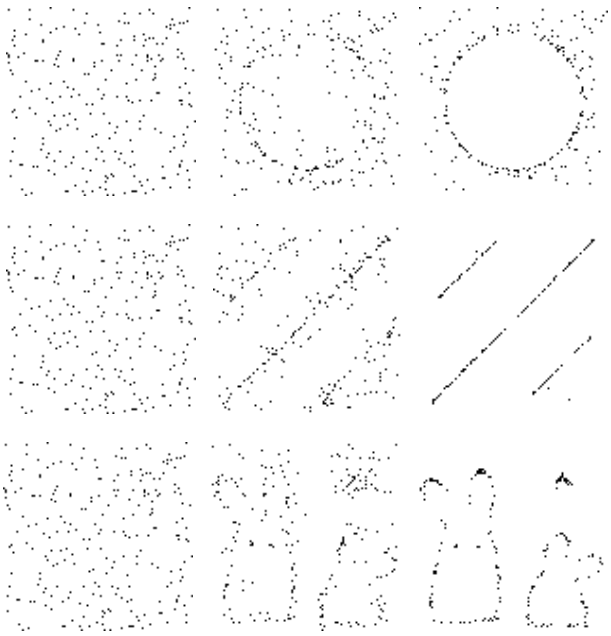
[2] Chii-Ruey Hwang. 1988. Simulated annealing: Theory and application. *Acta Applicandae Mathematica* 12 (1988), 108–111. https://doi.org/10.1007/BF00047572

[3] George Matejka, Justin und Fitzmaurice. 2017. Same Stats, Different Graphs: Generating Datasets with Varied Appearance and Identical Statistics through Simulated Annealing. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. Association for Computing Machinery, 1290–1294. https://doi.org/10.1145/3025453.3025912

[4] Sangit Chatterjee und Aykut Firat. 2007. Generating Data with Identical Statistics but Dissimilar Graphics: A Follow up to the Anscombe Dataset. *The American Statistician* 61, 3 (2007), 248–254. https://doi.org/10.2307/27643902

[5] Stephen Haslett und Kondaswamy Govindaraju. 2009. Cloning data: generating datasets with exactly the same multiple linear regression fit. *Australian & New Zealand Journal of Statistics* 51, 4 (2009), 499–503. https://doi.org/10.1111/j.1467-842X.2009.00560.x

[6] Kondaswamy Govindaraju und Stephen Haslett. 2008. Illustration of regression towards the means. *International Journal of Mathematical Education In Science & Technology* 39, 4 (2008), 544–550. https://doi.org/10.1080/00207390701753788



(g) Input (h) Interation: 20.000 (i) Iteration: 200.000

Abbildung 3: Modifizierungdsprozess einer Punktewolke in verschiedene Zielformen. Jeder der Datensätze hat bis auf zwei Nachkommastellen Genauigkeit die gleichen statistischen Eigenschaften ($mean_x = 51,42; mean_y = 50,04; stdDev_x = 28,76; stdDev_y = 29,75$).