

Generierung von visuell unterschiedlichen 2D Datensätzen mit ähnlichen statistischen Eigenschaften

Algorithm Engineering 2023 Project Paper

Eric Kaufmann
Friedrich Schiller University Jena
Germany
eric.kaufmann@uni-jena.de

Benjamin Schneg
Friedrich Schiller University Jena
Germany
benjamin.schneg@uni-jena.de

ABSTRACT

In dieser Veröffentlichung wurden, basierend auf initialen zweidimensionalen Datensätzen, neue Datensätze generiert, welche bis auf einen Fehler die gleichen statistischen Eigenschaften (Mittelwert, Standardabweichung und Korrelationskoeffizient) aufweisen. Dabei wurde ein paralleler Algorithmus entworfen, welcher mittels Simulated Annealing effizient zu einem Ergebnis kommt. Die originale Implementierung von Matejka und Fitzmaurice [1] wurde in Python geschrieben und ist daher ineffizient. Durch Verwendung der Programmiersprache C++ und einem parallelisierten Divide-and-Conquer-Ansatz wurde eine bis zu 390-fache Beschleunigung der Performance erreicht.

KEYWORDS

statistics, simulated annealing, mean, standard deviation, pearson coefficient, visualization

1 INTRODUCTION

In der Statistik ist die Analyse von Datensätzen maßgeblich. Bei besonders großen bzw. komplex strukturierten Datenmengen ist es häufig schwierig die Daten sinnvoll zu visualisieren. Demnach wird der Datensatz nur von Eigenschaften, wie dem Mittelwert oder der Standardabweichung, charakterisiert. Dabei können Datensätze, die sich visuell kaum ähnlich sind, dennoch nahezu identische statistische Eigenschaften aufweisen.

1.1 Background

Um Datensätze zu generieren, die bis auf einen bestimmten Fehler die gleichen statistischen Eigenschaften aufweisen, wird der Ansatz des Simulated Annealing verwendet. Die Idee ist, iterativ ein Problem näher an seine Optimallösung zu bringen. Es wird dabei akzeptiert, dass am Anfang größere Fehler zugelassen werden, diese Toleranz (Temperatur) jedoch über den Verlauf des Algorithmus nachlässt. Somit beschränkt sich der Algorithmus nicht nur auf lokale Optimallösungen, sondern deckt ein weites Optimierungsgebiet ab. Es kann daher eine globale Optimallösung erreicht werden.

1.2 Related Work

Die Idee basiert auf der Veröffentlichung *Same Stats, Different Graphs: Generating Datasets with Varied Appearance and Identical Statistics through Simulated Annealing* von Matejka und Fitzmaurice [1]. Dort wurde ein zweidimensionaler Datensatz mittels Simulated Annealing auf eine Zielstruktur transformiert. Sowohl der

initiale als auch der transformierte Datensatz wiesen dabei einen ähnlichen Mittelwert, eine ähnliche Standardabweichung und ähnlichen Korrelationskoeffizient auf. Ähnlich wurde dort mit einer Gleichheit auf von bis zu zwei Nachkommastellen definiert.

1.3 Our Contributions

In dieser Veröffentlichung wurde nun die Idee von Matejka und Fitzmaurice[1] aufgegriffen und optimiert. Anstatt den neuen Datensatz mittels Python zu generieren, wurde in dieser Arbeit versucht durch die Verwendung von C++ und OpenMP eine effizientere Lösung zu finden.

2 THE ALGORITHM

In diesem Abschnitt wird nun der Algorithmus erläutert. Dabei wird zunächst auf die Grundidee der Veröffentlichung von Matejka und Fitzmaurice[1] eingegangen und diese erläutert. Anschließend werden die Optimierungen an diesem Original beschrieben.

2.1 Algorithm description

Der Ablauf der Transformation in in Algorithmus 1 dargestellt. Als Input werden folgende Parameter erwartet:

- **iterations n :** Das ist die Gesamtanzahl der Iterationsschritte die der Algorithmus durchläuft. Eine größere Zahl bedeutet im Allgemeinen auch ein besseres Ergebnis.
- **inital dataset D :** Das ist der Initiale Datensatz, mit dem der Algorithmus startet. Die statistischen Eigenschaften dieses Datensatzes sollen auch versucht beibehalten zu werden.
- **target dataset T :** Der Zieldatensatz gibt die Form in Punkten an, in welche der initiale Datensatz transformiert werden soll. Dieser soll ähnliche statistische Eigenschaften, wie der initiale Datensatz, aufweisen.
- **temperature t :** Die Temperatur ist notwendig für die Simulated Annealing. Anhand des Iterationsschritts sowie der Temperatur kann bestimmt werden, ob ein Punkt zufällig akzeptiert wird, auch wenn wir keine Näherung an die Optimallösung haben.
- **error e :** Mittels des Errors legen wir die obere Schranke fest, um wie viel die statistischen Eigenschaften zweier Datensätze voneinander Abweichen dürfen.

Die Ausgabe des Algorithmus ist der transformierte Datensatz, welcher bis auf einen Fehler die gleichen statistischen Eigenschaften aufweist, jedoch visuell dem Targetdatensatz ähnelt.

Der Grundlegende Ablauf eines Iterationsschrittes ist dabei wie folgt: Zunächst wird ein zufälliger Punkt aus dem initialen Datensatz ausgewählt. Das Ziel ist, dass dieser so verschoben wird, dass

die statistischen Eigenschaften beibehalten werden, er jedoch näher an dem Targetdatensatz liegt. Demnach wird der Punkt zufällig in eine Richtung geshiftet. Dann wird kontrolliert, ob das Shiften den Punkt näher an den Targetdatensatz gebracht hat. Ist er nicht näher, hat er trotzdem die Chance zufällig akzeptiert zu werden. Die Wahrscheinlichkeit wird durch die Temperatur gesteuert. Wird er auch da nicht akzeptiert, dann wird er erneut zufällig geshiftet. Dies wird solange wiederholt, bis mindestens eine der beiden Bedingungen erfüllt ist. Nun wird kontrolliert, ob der Datensatz mit dem neuen geshifteten Punkt immernoch, bis auf einen festgelagerten Fehler, die gleichen statistischen Eigenschaften aufweist, wie der initiale Datensatz. Wenn nicht, dann wird der Punkt verworfen und der nächste Iterationsschritt wird gestartet. Wenn die Eigenschaften übereinstimmen, dann wird der neue geshiftete Punkt Teil des Datensatzes und der nächste Iterationsschritt wird gestartet.

Algorithm 1 Transform Dataset

```

function TRANSFORM(iterations  $n$ , initial dataset  $D$ , target dataset
 $T$ , temperature  $t$ , error  $e$ )
  for  $i = 1, \dots, n$  do
     $p \leftarrow \text{getRandomPoint}(D)$ 
    while  $\text{minDist}(p', T) \geq \text{minDist}(p, T)$  do
       $p' \leftarrow \text{randomShift}(p)$ 
      if  $\text{allowBreak}(i, t)$  then
        break
      end if
    end while
     $D' \leftarrow D \cup \{p'\} - \{p\}$  ▷ Ersetze  $p$  durch  $p'$  in  $D$ 
    if  $\text{sameStats}(D, D', e)$  then
       $D \leftarrow D'$ 
    end if
  end for
  return  $D$ 
end function

```

Zusammenfassend erreicht der Algorithmus, dass in fast jedem Iterationsschritt ein Punkt des ursprünglichen Datensatzes so verändert wird, dass die statistischen Eigenschaften beibehalten werden, jedoch die Form sich ein kleines Stück in Richtung des Targetdatensatz bewegt. Über eine große Anzahl von Iterationsschritten wurde ein neuer Datensatz mit ähnlichen statistischen Eigenschaften generiert.

2.2 Statistical Properties

Wie im Verfahren erläutert, werden einige statistische Eigenschaften von Datensätzen betrachtet. Dabei wurden folgende Parameter verwendet:

- Mittelwert
- Standardabweichung
- Korrelationskoeffizient

Im folgenden werden die Eigenschaften genauer beschrieben.

Mittelwert. Der Mittelwert bestimmt das arithmetische Mittel aller Punkte des Datensatzes für jede Dimension. Dieser wird dabei wie folgt berechnet:

$$\bar{D} = \begin{pmatrix} \bar{x} \\ \bar{y} \end{pmatrix} = \frac{1}{N} \sum_{i=1}^N \begin{pmatrix} x_i \\ y_i \end{pmatrix}.$$

In der visuellen Darstellung bietet der Mittelwert einen Punkt, um welchen sich die Daten erstrecken.

Standardabweichung. Die zweite statistische Eigenschaft ist die Standardabweichung. Sie ist ein Maß für die Streuung eines Datensatzes. Dabei wird in jeder Dimension der quadratische Abstand aller Punkte zum Mittelwert akkumuliert und gemittelt. Wie Berechnungsvorschrift ist wie folgt:

$$\sqrt{\text{Var}(D)} = \sqrt{\begin{pmatrix} \text{Var}(x) \\ \text{Var}(y) \end{pmatrix}} = \sqrt{\frac{1}{N-1} \sum_{i=1}^N \begin{pmatrix} (x_i - \bar{x})^2 \\ (y_i - \bar{y})^2 \end{pmatrix}}.$$

In der visuellen Betrachtung kann man mithilfe des Mittelwertes und der Standardabweichung bereits einiges aussagen. Somit kann nicht nur den Mittelpunkt der Daten betrachten, sondern auch wie weit sich die Daten von diesem Mittelpunkt in alle Dimensionen erstrecken. Um zuletzt noch die Form der Daten zu bestimmen wird die dritte statistische Eigenschaft verwendet.

Korrelationskoeffizient. Als Letztes wird der Korrelationskoeffizient herangezogen. Dieser zeigt den linearen Zusammenhang der x - und y -Werte eines Datensatzes auf. Das ist ein Wert im Bereich $[-1, 1]$. Bei einem großen absoluten Wert (1 oder -1) liegt eine vollständige Korrelation vor. Bildlich würde es bedeuten, dass die Daten auf einer Geraden ausgerichtet sind. Ein Wert von 0 würde bedeuten, dass wir eine perfekte symmetrische Verteilung der Daten auf den zwei Dimensionen haben, da die Daten unkorreliert sind. Beispielsweise würde ein zweidimensionaler Datensatz mit normalverteilten x - und y -Werten einen niedrigen absoluten Korrelationskoeffizienten generieren. Die mathematische Berechnungsvorschrift lautet wie folgt:

$$\text{Korr}(x, y) = \frac{\sum_{i=1}^N (x_i - \bar{x}) * \sum_{i=1}^N (y_i - \bar{y})}{N * \sqrt{\text{Var}(x)} * \sqrt{\text{Var}(y)}}.$$

Mittels dieser statistischen Eigenschaften lassen sich einige Aussagen über bildliche Darstellungen der Daten im zweidimensionalen Koordinatensystem treffen. Somit lässt sich vermuten, dass Datensätze, welche bis auf einen kleinen Fehler (z.B. bis auf die zweite Nachkommastelle genau), identische Mittelwerte, Standardabweichung und Korrelationskoeffizienten haben, sehr ähnlich in der bildlichen Darstellung aussehen. Wie sich in den Ergebnissen zeigen lässt, ist dies jedoch nicht zwangsläufig der Fall. Dabei muss erwähnt werden, dass der Korrelationskoeffizient nur den linearen Zusammenhang aufzeigt. Nichtlineare Abhängigkeiten sind jedoch häufig schwer korrekt abzubilden, sodass eine vereinfachte lineare Abbildung häufig bereits eine gute Approximation ist.

2.3 Optimizations

Der originale Programmiercode von Matejka und Fitzmaurice[1] ist in der Programmiersprache Python verfasst und sollte keine Lösung in minimaler Laufzeit erreichen. In dieser Veröffentlichung wurde nun jedoch versucht diesen Algorithmus zu optimieren. Durch

die Verwendung von C++ als Programmiersprache wird ein effizientes Speichermanagement, gute und einfache Möglichkeiten der Parallelisierung sowie Compileroptimierungen gewährleistet. Außerdem wird durch das Ausführen eines fertig kompilierten Programms bereits viel Zeit gespart, während bei Python noch ein Interpreter im Hintergrund Quellcode zur Laufzeit übersetzt. Allein durch diese Änderungen wird ein großer Performance-Gewinn erhofft. Um eine zusätzliche Optimierung zu erreichen, wurden ein paar pragmatische Änderungen vorgenommen, um eine Parallelisierung zu ermöglichen. So wurde sich auf das Berechnen von Punkt-zu-Punkt Abständen beschränkt, während das Original Punkt-zu-Gerade Abstände betrachtet. Die Shapes wurden in dieser Version durch eine Punktwolke approximiert und der Datensatz wurde an das Shape angenähert, indem Punkt-zu-Punkt Abstände des Datensatzes zur Shape-Punktwolke minimiert wurden. Dies ermöglicht eine einheitliche Funktion zur Abstandsberechnung und es müssen daher keine Linien definiert werden, mit denen Punkt-zu-Gerade Abstände berechnet werden.

Als nächstes wurde mittels OpenMP versucht, den seriellen C++ Programmiercode zu parallelisieren. Als Grundlage dafür wurde ein Divide-and-Conquer-Ansatz gewählt. Der initiale Datensatz wird demnach gleichmäßig in so viele Teile aufgeteilt, wie es Threads gibt (jedoch mit einem Maximum von 8). Anschließend wird der Iterationsschritt auf allen Teilen des Datensatzes parallel über alle Threads ausgeführt. Das bedeutet es werden somit gleichzeitig $num_threads = n$ (Anzahl der Threads) Punkte zufällig ausgewählt, geschiftet, und die Statistiken berechnet. Beim Aufteilen wurde noch mittels Padding versucht, dass die Daten für jeden Thread in eigenen Adressbereichen liegen. Dadurch soll false sharing vermieden werden. Die Laufzeit könnte somit idealisiert auf $\frac{1}{n}$ reduziert werden. Jedoch entstehen dabei neue Herausforderungen. Um wirklich unabhängig zwischen Threads rechnen zu können, kann nicht bei jedem Shift in einer Teilgruppe die Statistik des gesamten Datensatzes ausgewertet werden. Statistiken müssen also auch gruppenspezifisch berechnet werden. Es wird daher in jeder Gruppe der Beitrag zur Gesamtstatistik berechnet und versucht diesen Beitrag nach einem Shift gleich zu halten. Der Mittelwert, die Standardabweichung und der Korrelationskoeffizient wurden dafür so transformiert, dass sie sich aus einer Summe zusammensetzen. Wenn sich Teilsummen einer Summe nicht verändern, dann bleibt die Gesamtsumme auch gleich.

Der maximale Fehler, welcher entstehen darf wird also auf die einzelnen Gruppen aufgeteilt. Angenommen wir haben einen maximalen Fehler e_{ges} , welcher eingehalten werden muss, damit ein Punkt zum Datensatz hinzugefügt werden kann. Die Fehler der einzelnen Teilgruppen sind durch e_i definiert. Dabei muss gelten:

$$e_{ges} \geq \sum_{i=1}^n e_i$$

Um nun zu erreichen, dass alle Einschränkungen eingehalten werden wird der Fehler ebenfalls aufgeteilt:

$$e_i \leq \frac{e_{ges}}{n} \quad \forall i = 1, \dots, n$$

Somit kann gewährleistet werden, dass der Gesamtfehler immer eingehalten wird. Nachteil ist, dass somit ein an sich kleiner Fehler

nur akzeptiert werden kann, da diese Aufteilung davon ausgeht, dass alle Teilfehler maximal ausgereizt werden.

Problematisch an dem Divide-and-Conquer Ansatz ist zudem, dass eine Aufteilung in Teildatensätze den Freiheitsgrad einzelner Punkte beim Shiften einschränkt. Wenn ein Punkt in einer kleinen Gruppe verschoben wird, dann ist diese Veränderung sehr einflussreich auf die Gesamtstatistik der Gruppe. Ein Shift in einer großen Gruppe hingegen fällt kaum ins Gewicht der Gesamtstatistik. Shifts in kleineren Gruppen können also weniger häufig akzeptiert werden. Daher wurde in dieser Arbeit auch die Anzahl der maximalen Threads auf 8 beschränkt, weil das Ergebnis bei noch kleineren Gruppen zu schlecht ausfällt und das angestrebte Shape kaum zu erkennen war. Daher hat dieser Ansatz einen Trade-off von Geschwindigkeit und Genauigkeit zur Folge.

3 EXPERIMENTS

Im folgenden Abschnitt wird nun betrachtet, welche Auswirkungen die Optimierung des Algorithmus auf die Laufzeit hat.

Verglichen werden dabei folgende Fälle:

- der serielle Python Algorithmus aus der Publikation von Matejka und Fitzmaurice[1], welches die Implementierung ist, auf welcher die neue Umsetzung basiert.
- der Algorithmus umgeschrieben in C++ und seriell ausgeführt. Somit kann betrachtet werden, welchen Unterschied die Verwendung einer anderen Programmiersprache macht.
- der gleiche Algorithmus in C++, jedoch nun parallelisiert mit den Optimierungen aus Abschnitt 2.3 ausgeführt.

Die Experimente wurden dabei auf einem AMD Ryzen 7 6800 HS mit 8 Kernen und 16 Threads ausgeführt. Somit kann eine Vergleichbarkeit erreicht werden.

Experiment 1. Im ersten Experiment wurden alle drei Fälle über unterschiedlich vielen Iterationen ausgeführt. Dabei wurden 100'000, 200'000, 500'000 und 1'000'000 Iterationen als Ausgangslage verwendet. Die Zeiten wurden in Sekunden gemessen. Durch die große Anzahl der Iterationen wird gewährleistet, dass alle Algorithmen zu einer guten Lösung kommen. Bei dem Fall C++/parallel wurde die beste parallele Zeit verwendet, die erreicht werden konnte, ohne dass größere Verminderungen des Ergebnisses zur Folge gekommen sind. In diesem Fall wurden 8 Threads verwendet. Die Ergebnisse für dieses Experimente sind in Tabelle 1 zu erkennen.

Table 1: Vergleich der Algorithmen und Ausführungsart über 100'000, 200'000, 500'000 und 1'000'000 Iterationen. Alle Angaben in Sekunden.

Iterations	Python/serial	C++/serial	C++/parallel
100,000	132.00	1.56	0.34
200,000	208.00	3.71	0.76
500,000	520.00	12.79	2.69
1,000,000	1040.00	32.03	5.05

Allgemein lässt sich erkennen, dass allein die Verwendung einer Programmiersprache, wie C++ die Laufzeit enorm verkürzt. Durch

Compileroptimierungen, sowie der generellen effizienten Speicheroptimierung ist sie bei berechnungsintensiven Anwendungen klar im Vorteil. Somit brauchte unser serielle Algorithmus nur ca. 1% der Zeit, die der serielle Python Algorithmus benötigt hatte. Durch die zusätzliche Verwendung von OpenMP und die Aufteilung des Problems in Subprobleme konnte eine weitere Beschleunigung erreicht werden, sodass unser paralleler C++ Code ca. 4 bis 5 Mal schneller als der serielle Code ist. Bei 1'000'000 Iterationen ist er sogar 6 Mal so schnell. Im Vergleich zu dem seriellen Python Code konnten wir bei 1'000'000 Iterationen eine über 200-fache Beschleunigung erreichen. Bei 100'000 Iterationen sogar fast 390-fach.

Die Ergebnisse zeigen deutlich auf, wie stark der Einfluss auf die Laufzeit allein bei der Verwendung einer anderen Programmiersprache ist. Mit zusätzlicher Parallelisierung kann man die Laufzeit von über 17 Minuten auf 5 Sekunden reduzieren.

Experiment 2. Im zweiten Experiment wurde nun der parallele C++ Algorithmus aufgegriffen und mit unterschiedlich vielen Threads ausgeführt. Auch hier wurden erneut 100'000, 200'000, 500'000 und 1'000'000 Iterationen als Ausgangslage verwendet. Mithilfe der zusätzlichen Benutzung der Threads soll betrachtet werden, in wie weit ein Speedup erreicht werden kann. Erwartet wird, dass der inhärent serielle Anteil, also der Bestandteil des Algorithmus, welcher schlecht bzw. gar nicht parallelisierbar ist, niedrig ist. Somit sollte die Verwendung mehrerer Thread zu einem größeren Performance-Gewinn führen. Die Ergebnisse des zweiten Experiments sind in Tabelle 2 aufgelistet.

Table 2: Vergleich der des parallelen C++ Algorithmus mit unterschiedlich vielen Threads über 100'000, 200'000, 500'000 und 1'000'000 Iterationen. Alle Angaben in Sekunden.

Iterations	2 Threads	4 Threads	8 Threads
100,000	0.84	0.44	0.34
200,000	2.07	1.19	0.76
500,000	7.09	3.87	2.69
1,000,000	18.34	10.54	5.05

Dabei lässt sich gut erkennen, dass das Problem mit mehr Threads, die parallel ausgeführt werden, schneller die Iterationen abarbeiten kann. Die Anzahl der Threads hat somit einen Einfluss auf die Laufzeit des Algorithmus. Dabei ist der Geschwindigkeitszuwachs anscheinend kaum von der Anzahl der Iterationen abhängig. Das bedeutet, dass über alle Iterationen ein Geschwindigkeitszuwachs von 1.5 bis 2 bei Verdoppelung der Threads angenommen werden kann. Dabei muss beachtet werden, dass die Lösung des Problems bei vielen Threads an Genauigkeit verliert. Das liegt an den fehlenden Freiheitsgraden beim Shiften. Somit werden, um ähnlich gute Ergebnisse zu erreichen, mehr Iterationen benötigt, je mehr Threads verwendet werden. Demnach muss ein Trade-off aus Anzahl der Threads und Iterationen gefunden werden.

Somit bestätigen die Experimente die Ziele eine effizientere Generierung eines Datensatzes mit ähnlichen statistischen Eigenschaften zu ermöglichen.

4 CONCLUSIONS

Zusammenfassend wurde in dieser Veröffentlichung gezeigt, dass man effizient aus zweidimensionalen Datensätzen neue Datensätze erstellen kann mit ähnlichen statistischen Eigenschaften. Dabei wurde experimentell gezeigt, dass allein die Verwendung einer anderen effizienteren Programmiersprache, wie C++, es erlaubt die Laufzeit enorm zu reduzieren. Durch Parallelisierung mit dem Divide-and-Conquer-Ansatz kann die Laufzeit erneut, je nach Anzahl der Threads, geviertelt werden. Dabei ist dieser Performance-Gewinn stetig über eine variable Anzahl an Iterationen.

REFERENCES

- [1] Justin Matejka and George 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*. 1290–1294.