Generierung von visuell unterschiedliche 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 entwurfen, welcher mittels Simmulated Annealing effizient zu einem Ergebnis kommt. Der originale Implementierung von Matejka und Fitzmaurice [1] wurde ineffizient mittels Python implementiert. Durch Verwendung der Programmiersprache C++ und einem parallelisierten Divide-and-Conquer-Ansatz wurde eine bis zu 390-fache Beschleunigung der Performance erreicht.

KEYWORDS

statistics, simmulated annealing, mean, standard deviation, pearson coefficient

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, klassifiziert. 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 aufweist, wird der Ansatz des Simmulated Annealing verwendet. Die Idee ist dabei iterativ ein Problem näher an seine Optimallösung zu bringen. Es wird dabei akzeptiert, dass man am Anfang größere Fehler zulässt, diese Toleranz (Temperatur) jedoch über die Interationen nachlässt. Das heißt, das nach jeder Iteration die erlaubte Veränderung gedämpft wird, sodass man exakter die Lösung approximieren kann.

1.2 Related Work

Die Idee basiert auf 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 Datensätze mittels Simmulated Annealing auf eine Zielstruktur transformiert. Sowohl der initiale als auch der transformierte Datensatz wiesen dabei einen ähnlichen Mittelwert, eine ähnliche Standwardabweichung und ähnlichen Korrelationskoeffizient auf. Ähnlich wurde dort mit einer Gleichheit auf von bis zu zwei Nachkommastellen definiert.

1.3 Our Contributions

In dieser Veröffentlichen wurde nun die Idee von Matejka und Fitzmaurice[1] aufgegriffen und optimiert. Anstatt den neuen Datensatz mittels Python zu gegerieren, 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 werdnen 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 Ergbenis.
- **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 Simmulated Annealing. Anhand des Iterationsschritts sowie der Temperatur kann bestimmt werden, ob ein Punkt zufällig akzeptiert wird, auch wenn wir keine Nährung 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 Agorithmus 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 geshifted. 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 geshifted. 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 Fehler, die gleichen statistischen Eigenschaften aufweist, wie der initiale Datensatz. Wenn nein, dann wird der Punkt verworfen und der nächste Iterationsschritt wird gestaret. Wenn die Eigenschaften übereinstimmen, dann wird der neue geshiftet Punkt Teil des Datensatzes und der nächste Iterationsschritt wird gestartet.

Algorithm 1 Transform Dataset

```
function TRANSFORM(iterations n, inital dataset D, target dataset
T, temperature t, error e)
    for i = 1, ..., n do
        p ← getRandomPoint(D)
        while minDist(p', T) ≥ minDist(p, T) do
            p' ← randomShift(p)
            if allowBreak(i, t) then
                break
            end if
        end while
        D' ← D ∪ {p'} − {p}          ▷ Ersetze p durch p' in D
        if sameStats(D, D', e) then
            D ← D'
        end if
    end for
    return D
end function
```

Zusamenfassend erreicht der Algorithmus, dass in fast jedem Interationsschritt 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 wiefolgt berechnet:

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

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:

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

In der visuellen Betrachung 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 Dimesionen erstreckt. Um zuletzt noch die Form der Daten zu betimmen wird die dritte statistische Eigenschaft verwendet.

Korrelationskoeffizient. Als Letzes 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:

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

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 Korellationskoeffizienten haben, sehr ähnlich in der bildlichen Darstellung aussehen. Wie sich in den Ergebnissen zeigen lässt, ist dies jedoch nicht immer 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. Allein duch diese Änderungen wird ein großer Performance-Gewinn erhofft. Um einen zusätzliche Optimierung zu erreichen wurde die Berechnung der minimalen Distanz von einem Punkt zum Targetdatensatz abgeändert. In der Originalversion wird der Targetdatensatz durch Linien definiert. Der kürzeste Abstand wird somit durch die Berechnung der Distanz von Punkt zu Linie durchgeführt. In der optimierten Version werden die Linien durch Punkte diskretisiert. Somit wird die Distanz von Punkt zu Punkt bestimmt, welches effizienter zu berechnen ist. Zusätzlich wird ermöglicht, dass durch eine größere Diskretisierung, d.h. weniger Punkte pro Linie, die Laufzeit zur Berechnung der minimalen Distanz weiter reduziert werden kann. Dabei muss jedoch beachtet werden, dass zu wenige Punkte zu einem schlechteren Ergebnis führen. Somit muss ein sinnvoller Trade-of erreicht 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. 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, geshiftet, und die Statistiken berechnet. Somit könnte die Laufzeit idealisiert auf $\frac{1}{n}$ reduziert werden. Jedoch entstehen Probleme. Durch die Aufteilung des Datensatzes in Teildatensätze ist es schwierig die statistischen Eigenschaften zu berechnen, da sowohl für den Mittelwert, als auch der Standardabweichung alle Punkte des Datensatzes benötigt werden. Außerdem werden die statistischen Eigenschaften des gesamten Datensatzes benötigt, um eine Entscheidung treffen zu können, ob ein geshifteter Punkt zum Datensatz hinzugefügt werden soll oder nicht.

Um dieses Problem zu lösen spalten wir den maximalen Fehler, welcher entstehen kann auf. Angenommen wir haben einen maximalen Fehler e_{ges}, welcher eingehalten werden muss, dass ein Punkt zum Datensatz hinzugefügt werden kann. Die Fehler der einzelnen Teildatensätze 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:

$$e_i \leq \frac{e_{ges}}{n} \; \forall i = 1, \ldots, n$$

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.

Das nächste Problem ist die Berechnung des Teilfehlers, denn bei der Berechnung des Mittelwertes und der Standardabweichung wird durch die Anzahl der Daten N durch die Anzahl der Daten des Teildatensatzes $\frac{N}{n}$ ersetzt. Somit werden aber auf die Freiheitsgrade eingeschränkt, d.h. die statistischen Eigenschaften sind ungenauer und fehleranfälliger.

Insgesamt besteht das Problem, dass durch die Aufteilung das Problem schwieriger wird, da der Fehler stärker beschränkt wird. Dennoch kann durch die Aufteilung des Problems in Subprobleme sinnvoll sein, da so mehr Iterationen in gleicher Zeit durchgeführt werden können.

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 Ergenisse 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 Experiemnt 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 Lösung des Problems bei vielen Threads and Genauigkeit verliert. Das liegt an den fehlenden Freiheitsgraden. Somit werden, um ähnlich gute Ergebnisse zu erreichen, mehr Iterationen benötigt, je mehr Threads verwendet werden. Demnach muss ein Trade-off auf 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 einen enormen Gewinn der Laufzeit zu erreichen. 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.