DHBW Ravensburg
Data Science und Künstliche Intelligenz
WDS123

Projektbericht zum Thema:

Sortieralgorithmen – Implementierung eines Sortieralgorithmus mit Beispieldatensatz aus der Formel 1

Vorgelegt von :
Carl Jonas Lederer
Westenstraße 11
83376 Truchtlaching

Matrikelnummer:
1405269

Abgabedatum:
30.05.2024

Prüfer:
Steffen Merk

INHALT

Einleitung

Sortieralgorithmen spielen in der Informatik eine zentrale Rolle und finden in zahlreichen Anwendungen zur Analyse und Informationsgewinnung Verwendung. In der folgenden Projektarbeit werden Sortieralgorithmen im Kontext einer konkreten Problemstellung behandelt. Zunächst werden die fünf ausgewählten verschiedenen Sortieralgorithmen Selection Sort, Bubble Sort, Quick Sort, Insertion Sort und Merge Sort beschrieben und differenziert. Der geeignetste Sortieralgorithmus zum Lösen der Problemstellung wird anhand von verschiedenen Kriterien ausgewählt.

Zur Implementierung des Algorithmus wurde ein Datensatz mit Daten über den Reifenverbrauch von Formel-1-Fahrern auf verschiedenen Strecken sowie deren Kosten erstellt. Außerdem beinhaltet der Datensatz verschiedene Strategien der Rennställe auf den jeweiligen Strecken sowie die Endplatzierungen des Fahrers. Die Formel 1 ist die größte und bekannteste Rennserie der Welt und erregt Jahr für Jahr weltweit viel Aufmerksamkeit. Da in der Formel 1 oft Zehntelsekunden über Sieg oder Niederlage entscheiden, ist sie ein Sport in dem Daten eine zentrale Rolle spielen. Die Implementierung des Codes soll die Reifenkosten zunächst nach Fahrern gruppieren und anschließend absteigend nach Reifenkosten sortieren. Zuletzt wird die Auswahl und Implementierung des Sortieralgorithmus evaluiert und gegebenenfalls Anpassungen vorgenommen.

Beschreibung der Sortieralgorithmen

Selection Sort

Der Selection Sort Sortieralgorithmus ist ein einfach zu implementierender Algorithmus, der auf dem Vergleichen von Werten basiert. Beim Selection Sort Algorithmus wird das eingegebene Array mehrfach auf den kleinsten Wert durchsucht. Im ersten Durchlauf wird der kleinste gefundene Wert - vorausgesetzt es ist eine aufsteigende Sortierung gewünscht - an den Index[0] des Arrays gesetzt. Durch das Inkrementieren des Startindexes der Suche werden im folgenden Suchdurchlauf nur die noch nicht sortierten Elemente durchsucht. Dieser Prozess wird so lange wiederholt bis das gesamte Array sortiert ist.

Selection Sort ist ein nicht stabiler Sortieralgorithmus, da es vorkommen kann, dass bei der Sortierung der Elemente die relative Position von gleichwertigen Elementen vertauscht wird. Dadurch, dass der Algorithmus das gegebene Array für jeden zusätzlichen Wert erneut komplett durchsucht, ergibt sich eine exponentiell wachsende Zeitkomplexität O(n^2). Bei kleinen Datensätzen ist Selection Sort performant; mit steigender Anzahl der Daten sinkt die Effizienz aufgrund der Zeitkomplexität stark. Die Speicherkomplexität beträgt hingegen aufgrund der In-place Verarbeitung der Daten O(1). Da sich Selection Sort nur für kleine Datenmengen eignet, findet er in der Praxis bei kleinen Problemstellungen Anwendung.

Bubble Sort

Der Bubble Sort Algorithmus ist ebenfalls ein einfach zu implementierender Algorithmus, der darauf basiert, wiederholt benachbarte Elemente zu untersuchen und diese - sollten sie falsch sortiert sein - zu vertauschen. Bei jedem Durchlauf durch das zu untersuchende Array wird das größte Element an das Ende des Arrays sortiert. Dieses Vorgehen wird wiederholt, bis keine Vertauschung der Elemente mehr erforderlich ist und das gesamte Array sortiert ist.

Bubble Sort ist aufgrund seiner Zeitkomplexität von O(n^2) ebenfalls nicht für größere Datenmengen geeignet. Für jedes zusätzliche Element muss der Algorithmus exponentiell mehr Vergleiche der Elemente durchführen. Da auch Bubble Sort - ähnlich wie Selection Sort - ein in-place Algorithmus ist und eine konstante Menge an zusätzlichem Speicher benötigt, ist die Speicherkomplexität auch hier O(1). Im Gegensatz zu Selection Sort behält Bubble Sort die ursprüngliche Reihenfolge gleichwertiger Elemente bei. Bubble Sort findet in der Praxis selten Anwendung aufgrund der Unzweckmäßigkeit für große Datenmengen.

Insertion Sort

Insertion Sort ist ein weiterer Sortieralgorithmus, der eine andere Ansatz der Sortierung verwendet. Der Algorithmus ist vergleichbar mit dem Sortieren von Spielkarten. Hierbei wird das zu sortierende Array in zwei Hälften - einem Sortierten und einem Unsortierten Teil - unterteilt. Die Elemente des unsortierten Teils werden mehrfach durchlaufen und der jeweils größte gefundene Wert in den Sortierten Teil eingefügt. Das Einfügen des Elements geschieht ebenfalls mit einem Durchlauf durch den sortierten Teil, wobei jedes sortierte Element mit dem Einzufügenden verglichen wird. Ist das unsortierte Element kleiner als das bereits sortierte Element, wird dieses vor dem bereits sortierten Element eingefügt. Hierfür muss außerdem der Index aller sortierten Elemente, welche größer sind, um Eins inkrementiert werden. Die Sortierung ist abgeschlossen, wenn sich alle Elemente des ursprünglichen Arrays im sortierten Array befinden.

Die Zeitkomplexität von Insertion Sort kann stark variieren. Im besten Fall (best case) - einem bereits vollständig sortiertem Array - beträgt die Zeitkomplexität O(n), da hierbei das Array nur einmal durchlaufen werden muss. Im schlechtesten Fall (worst case) - einem Array welches komplett falschherum sortiert ist - liegt die Zeitkomplexität bei O(n^2), da hierbei für jedes Element die maximale Anzahl an Durchläufen durchgeführt wird. Da der Insertion Sort Algorithmus ähnlich wie Bubble Sort und Selection Sort ein in-place Algorithmus ist, beträgt die Speicherkomplexität O(1). Die ursprüngliche Reihenfolge gleichwertiger Elemente wird bei Insertion Sort ebenfalls beibehalten. Aufgrund seiner guten Performance auf kleinen, bereits teilweise sortierten Datensätzen, wird Insertion Sort in der Praxis vor allem angewendet, wenn nach und nach Elemente in eine bereits sortierte Liste eingefügt werden müssen. Für große Datensätze, die zudem unsortiert sind, steigt die Zeitkomplexität exponentiell an, was diesen Algorithmus für diese Art von Datensätzen unattraktiv macht.

Quick Sort

Quick Sort ist im Vergleich zu den vorherigen Sortieralgorithmen aufgrund seines Aufbaus effizienter. Der Quick Sort Algorithmus bedient sich des Divide-and-Conquer-Prinzips. Dies bedeutet, dass die Sortierung in mehrere Teilschritte unterteilt wird und durch Rekursion zunächst Teilsortierungen durchgeführt werden. Zunächst wird ein Pivot-Element ausgewählt, wobei Dieses frei wählbar ist. Anschließend wird das ursprüngliche Array an der Stelle des Pivot-Elements geteilt und alle Elemente des Arrays werden mit dem Pivot-Element verglichen. Alle Elemente, die kleiner als das Pivot-Element sind, befüllen das eine („linke“) Array; alle Werte die größer sind, das andere („rechte“) Array. Durch Rekursion wird dieser Vorgang dann solange wiederholt - das heißt die Arrays werden so lange geteilt und sortiert - bis das gesamte

Array sortiert ist. Hierbei ist anzumerken, dass immer zunächst das linke und damit das Array mit den kleineren Elementen sortiert wird. Erst wenn der komplette linke Teil sortiert ist, wird der rechte Teil sortiert.

Ähnlich wie bei Insertion Sort kann die Zeitkomplexität von Quick Sort stark variieren. Entscheidend hierbei ist die Auswahl des Pivot-Elements. Teilt das Pivot-Element den Datensatz genau in der Mitte, ist der Algorithmus mit einer Zeitkomplexität von O(n log n) sehr performant (best case). Bei zufälliger Auswahl des Pivot-Elements kann es vorkommen, dass das Pivot-Element dem größten oder kleinsten Element des Arrays entspricht. Hierbei erfolgt die anschließende Sortierung nach dem Prinzip des Insertion Sort Algorithmus (worst case). Analog dazu beträgt die Zeitkomplexität dann O(n^2). Im Durchschnitt (average case) werden die Daten durch das Pivot-Element in zwei ähnlich große Hälften geteilt. In diesem Fall beträgt die Zeitkomplexität dann ebenfalls O(n log n). Da Quick Sort ebenfalls ein in-place Sortieralgorithmus ist und keine Datenstrukturen außerhalb der Sortierung anlegt, ist auch hier die Speicherkomplexität O(1). Da die ursprüngliche Reihenfolge der gleichwertigen Elemente wird unter Umständen nicht beibehalten wird, ist Quick Sort ein instabiler Sortieralgorithmus.

In der Praxis findet Quick Sort auch heute noch viel Verwendung. Er eignet sich durch seine Zeitkomplexität von O(n log n) im Best und Average Case dazu, große Datenmengen zu sortieren.

Merge Sort

Das Funktionsprinzip von Merge Sort ähnelt dem Quick Sort Algorithmus. Auch Merge Sort bedient sich dem Divide-and-Conquer-Prinzip und arbeitet mit Rekursion. Zudem ist er sehr effizient und gleichzeitig stabil. Im Merge Sort Algorithmus wird zunächst das zu sortierende Array in zwei Hälften - eine linke und eine rechte Hälfte - geteilt. Dieser Vorgang wird rekursiv wiederholt, bis jedes Teilarray nur noch jeweils ein Element enthält. In einer weiteren Methode (im Folgenden „merge()“) werden die einzelnen Arrays dann verglichen und in einem neuen Array sortiert und zusammengefügt. Durch das rekursive Verhalten wird zunächst der gesamte linke Teil, also die Hälfte aller Daten, sortiert; anschließend der rechte Teil. Nach Abschluss der letzten Rekursionsschleife ist das gesamte Array sortiert. Durch die Aufteilung der Sortierung in mehrere Teilprobleme, besteht die Möglichkeit die Ausführung des Quell Codes zu parallelisieren.

Durch den Aufbau garantiert der Merge Sort Algorithmus eine konstante Zeitkomplexität O(n log n). Dieses Alleinstellungsmerkmal macht Merge Sort im Vergleich zu den bisher behandelten Sortieralgorithmen zu dem effizientesten Algorithmus bei großen Datenmengen. Des Weiteren hat Merge Sort eine Speicherkomplexität von O(n), diese ist aufgrund der

temporären Speicherung von Arrays deutlich höher als bei den bisherigen Sortieralgorithmen. Die höhere Speicherkomplexität ist nur ein schwacher Nachteil, da die heutige Hardware in der Regel über ausreichend Ressourcen verfügt. Außerdem ist der Merge Sort wie Bubble Sort und Insertion Sort ein stabiler Sortieralgorithmus.

Auswahl eines geeigneten Sortieralgorithmus

Die Auswahl eines geeigneten Sortieralgorithmus zur Implementierung erfolgt anhand verschiedener Auswahlkriterien. Zum einen soll der Algorithmus in der Lage sein, eine Milliarde Einträge zu sortieren; zum anderen soll es möglich sein, die Ausführung des Codes zu parallelisieren, das heißt das Teile des Codes parallel ausgeführt werden können, ohne das Endergebnis zu beeinflussen.

Aufgrund der vorgegebenen Größe des Datensatzes eignen sich die Sortieralgorithmen mit einer Zeitkomplexität von O(n^2) nicht. Aus diesem Grund werden die Sortieralgorithmen Selection Sort und Bubble Sort nicht weiter berücksichtig. Da Insertion Sort nur im best case Szenario - welches in der Praxis selten vorkommen dürfte - eine Zeitkomplexität von O(n) hat, ist dieser ebenfalls ungeeignet. Quick Sort und Merge Sort haben eine Zeitkomplexität von bis zu O(n log n), wobei nur Merge Sort diesen hervorragenden Wert auch konstant bieten kann. Um die Voraussetzung der Parallelisierung der Ausführung zu gewährleisten, muss der ausgewählte Sortieralgorithmus das Divide-and-Conquer-Prinzip implementieren. Dies trifft ebenfalls nur auf Quick Sort und Merge Sort zu.

Aus diesen Erkenntnissen lässt sich ableiten, dass der geeignetste Sortieralgorithmus für die gegebene Problemstellung Merge Sort ist. Merge Sort ist mit der Zeitkomplexität von O(n log n) der schnellste und gleichzeitig zuverlässigste Sortieralgorithmus von allen. Außerdem lässt er eine Parallelisierung des Codes zu. Ein weiterer Vorteil im Vergleich zu Quick Sort ist die Stabilität von Merge Sort. In der Praxis kann dies unter Umständen ein wichtiger Faktor sein.

Quellcode Dokumentation

Klasse Aggregator

```
public class Aggregator {
    public ArrayList<AggregatedRow> aggregate(ArrayList<Row> rows) {

        ArrayList<AggregatedRow> aggregatedRows = new ArrayList<>();

        for (Row row : rows) {
            AggregatedRow existingRow = findAggregatedRow(aggregatedRows, row.driver);
            if (existingRow != null) {
                existingRow.totalCostTyre += row.totalCostTyre;
            } else {
                aggregatedRows.add(new AggregatedRow(row.driver, row.totalCostTyre));
            }
        }
        return aggregatedRows;
    }

    private AggregatedRow findAggregatedRow(ArrayList<AggregatedRow> aggregatedRows, String
driver) {
        for (AggregatedRow a : aggregatedRows) {
            if (a.driver.equals(driver)) {
                return a;
            }
        }
        return null;
    }
}
```

Klasse Row

```
public class Row {
    public String driver;
    public double totalCostTyre;

    public Row(String driver, double totalCostTyre) {
        this.driver = driver;
        this.totalCostTyre = totalCostTyre;
    }
}
```

Klasse AggregatedRow

```
public class AggregatedRow {
    public String driver;
    public double totalCostTyre;

    public AggregatedRow(String driver, double totalCostTyre){
        this.driver = driver;
        this.totalCostTyre = totalCostTyre;
    }
}
```

Klasse Sorter

```
public class Sorter {
    public void sort(ArrayList<AggregatedRow> rows) {
        mergeSort(rows);
    }

    private static void mergeSort(ArrayList<AggregatedRow> rows) {
        int length = rows.size();
        if (length < 2) {
            return;
        }
        int middleIndex = length / 2;
        ArrayList<AggregatedRow> leftSide = new ArrayList<>(rows.subList(0, middleIndex));
        ArrayList<AggregatedRow> rightSide = new ArrayList<>(rows.subList(middleIndex,
length));
        mergeSort(leftSide);
        mergeSort(rightSide);
        merge(rows, leftSide, rightSide);
    }

    private static void merge(ArrayList<AggregatedRow> rows, ArrayList<AggregatedRow>
leftSide, ArrayList<AggregatedRow> rightSide) {
        int leftIndex = 0;
        int rightIndex = 0;
        for (int i = 0; i < rows.size(); i++) {
            if (leftIndex >= leftSide.size()) {
                rows.set(i, rightSide.get(rightIndex));
                rightIndex++;
                continue;
            } else if (rightIndex >= rightSide.size()) {
                rows.set(i, leftSide.get(leftIndex));
                leftIndex++;
                continue;
            } else if (leftSide.get(leftIndex).totalCostTyre >=
rightSide.get(rightIndex).totalCostTyre) {
```

```
                rows.set(i, leftSide.get(leftIndex));
                leftIndex++;
            } else {
                rows.set(i, rightSide.get(rightIndex));
                rightIndex++;
            }
        }
    }
}
```

Schluss

In dieser Arbeit wurden zunächst fünf verschiedene Sortieralgorithmen vorgestellt. Anschließend wurde einer dieser Sortieralgorithmen anhand von zwei Kriterien ausgewählt, um das Sortierproblem des Formel 1 Datensatzes zu lösen. Aufgrund der Vorgabe, dass der Datensatz eine Milliarde Einträge und die Möglichkeit der Parallelisierung bieten soll, war eine Implementierung eines Merge Sort Algorithmus am Sinnvollsten. Im vorangegangenen Quell Code wurde der Merge Sort Algorithmus implementiert. Zur Erfolgsprüfung wurde zusätzlich eine Main-Klasse mit Daten implementiert, welche nicht Bestandteil dieser Arbeit ist. Der implementierte Algorithmus hat die gegebene Aufgabenstellung, die Fahrer absteigend nach Reifenkosten zu sortieren, stabil und zuverlässig ausgeführt.

Eigenständigkeitserklärung

Ich versichere hiermit, dass ich die vorliegende Projektarbeit mit dem Thema „Sortieralgorithmen – Implementierung eines Sortieralgorithmus mit Beispieldatensatz aus der Formel 1“ selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ravensburg, den 27.05.2024

_____

Carl Jonas Lederer