Sortieralgorithmen Satisfactory

PROJEKTBERICHT

des Studienganges Data Science und Künstliche Intelligenz

an der Dualen Hochschule Baden-Württemberg Ravensburg

von

Jannes Kurzke

Abgabedatum

30. Mai 2024

Bearbeitungszeitraum	4 Wochen
Matrikelnummer, Kurs	1454387, WDS123
Dualer Partner	Airbus Defence und Space, Immenstaad

AIRBUS

DHBW
Duale Hochschule
Baden-Württemberg

Inhaltsverzeichnis

Einleitung

Das Computerspiel Satisfactory scheint zunächst recht einfach zu sein. Zu Beginn des Spiels ist der Spieler hauptsächlich damit beschäftigt, Eisenerz abzubauen und es im Ofen zu schmelzen. Im Laufe der Zeit schaltet der Spieler jedoch eine Vielzahl von Rezepten frei, und die Komplexität des Spiels nimmt rapide zu. Es ist möglich, dass für die Herstellung fortschrittlicher Gegenstände wie Supercomputer dutzende Unterbestandteile benötigt werden.

Datensatz

Um das Spielererlebnis zu verbessern, wurden alle bislang freigeschalteten Rezepte in einer Datenbank erfasst. Jeder Datensatz in der Datenbank repräsentiert ein einzelnes Rezept. Die erste Spalte gibt das Ziel-Item des Rezepts an, gefolgt von Spalten zur Dauer und Produktionsrate des Rezepts sowie zur genutzten Maschine. Da einige Rezepte mehrere Outputs haben können, werden zwei mögliche Outputs aufgelistet, wobei der erste Output immer das Ziel-Item darstellt. Mit zunehmender Komplexität der Rezepte steigt auch die Anzahl der benötigten Input-Items. Daher werden im Datensatz vier mögliche Inputs mit dazugehörigen Produktionsmengen und -raten aufgelistet.

Problemstellung

Im Verlauf des Spiels werden zunehmend mehr Varianten für einzelne Rezepte freigeschaltet, die unterschiedliche Produktionsraten aufweisen können. Es kann vorkommen, dass im Datensatz ein Item mit mehreren Rezepten aufgeführt wird. Da die Produktionsstrecke nun jedoch auf die höchstmögliche Produktionsrate pro Minute optimiert werden soll, filtere ich mittels der Aggregator Klasse nur diejenigen Rezepte heraus, die die höchste Produktionsrate pro Minute haben.

Anschließend werden die verbleibenden Rezepte absteigend nach der Produktionsrate sortiert. Dies ermöglicht einen besseren Überblick über das gesamte Repertoire an Rezepten.

Sortieralgorithmen

Voraussetzungen

Die Informatik kennt heutzutage eine beträchtliche Anzahl an Sortieralgorithmen und jeder von Ihnen hat einen Platz und eine Zeit. Einige sind besonders einfach nachzuvollziehen und zu implementieren. Andere hingegen sind besonders effizient und haben einen niedrigen Speicherbedarf. Zur Evaluation des passenden Algorithmus, werde ich mich auf die Performance im schlechtesten Fall beziehen, da dieser häufig zu einem Bottle Neck wird.

Bei meinem Algorithmus ist besonders die Performance relevant, da mein Programm auf einen Datensatz von einer Milliarde Datenpunkten anwendbar sein sollte. Um die Geschwindigkeit zu erhöhen wäre es von Vorteil, wenn eine parallele Verarbeitung der Sortierung möglich wäre, um die Last auf mehrere Threads zu verteilen. Ebenso wird ein Fokus auf den Speicherbedarf gelegt, da dieser sonst schnell Überhand nehmen kann bei so vielen Datenpunkten.

Selection Sort

Selection Sort ist ein vergleichsweise einfacher, jedoch instabiler, In-Place-Sortieralgorithmus. Der Algorithmus arbeitet nach dem Prinzip, den unsortierten Teil der Liste zu durchlaufen und den Index des kleinsten (oder größten) Elements zu ermitteln. Dieses Element wird anschließend mit dem ersten Element des unsortierten Abschnitts vertauscht und das Element ist somit sortiert. Diese Schritte werden wiederholt, bis die gesamte Liste sortiert ist.

Dieser Algorithmus scheidet aufgrund seiner hohen Zeitkomplexität von $O(N^2)$ bei großen Datensätzen aus. Dennoch weist er dank seiner In-Place-Eigenschaft einen sehr geringen Speicherbedarf von $O(1)$ auf.

Bubble Sort

Bubble Sort ist einer der bekanntesten stabilen In-Place-Sortieralgorithmen. Der Algorithmus arbeitet nach dem Prinzip, die Liste von vorne nach hinten zu durchlaufen und dabei jedes Element mit seinem Nachfolger zu vergleichen. Ist das aktuelle Element größer (oder kleiner) als sein Nachfolger, werden die beiden Elemente vertauscht. Danach wird zum nächsten Element weitergegangen. Dieser Vorgang wird wiederholt, bis das Ende des unsortierten Teils der Liste erreicht ist. Die größten (oder kleinsten) Elemente steigen somit langsam innerhalb der Liste auf, bis sie an ihrem richtigen Platz sind. Wenn die Liste vollständig durchlaufen wurde, ohne dass Vertauschungen vorgenommen wurden, ist die Liste vollständig sortiert.

Dieser Algorithmus scheidet aufgrund seiner hohen Zeitkomplexität von $O(N^2)$ bei großen Datensätzen aus. Dennoch weist er dank seiner In-Place-Eigenschaft einen sehr geringen Speicherbedarf von $O(1)$ auf.

Insertion Sort

Insertion Sort ist ein weiterer stabiler In-Place-Sortieralgorithmus, der im Vergleich zu den vorherigen Algorithmen etwas komplexer ist. Der Algorithmus funktioniert nach dem Prinzip, die Liste von vorne nach hinten zu durchlaufen und jedes Element mit dem vorherigen zu vergleichen. Wenn das aktuelle Element kleiner (oder größer) ist als das vorherige, werden sie vertauscht. Dieser Vorgang wird wiederholt, indem man rückwärts durch die bereits sortierten Elemente geht und bei Bedarf Vertauschungen vornimmt. Wenn keine weiteren Vertauschungen mehr stattfinden, springt der Algorithmus zum Anfang des unsortierten Teils zurück, um den nächsten Durchlauf zu beginnen. Sobald das Ende der Liste erreicht ist, ist die Liste vollständig sortiert.

Dieser Algorithmus scheidet aufgrund seiner hohen Zeitkomplexität von $O(N^2)$ bei großen Datensätzen aus. Dennoch weist er dank seiner In-Place-Eigenschaft einen sehr geringen Speicherbedarf von $O(1)$ auf.

Quick Sort

Quick Sort ist ein schneller, rekursiver, jedoch instabiler Sortieralgorithmus. Er funktioniert nach dem Prinzip, die Liste in zwei Teile zu teilen, basierend auf einem sogenannten Pivot-Element. Alle Elemente, die kleiner als das Pivot-Element sind, werden auf die linke Seite verschoben, während die größeren Elemente auf die rechte Seite verschoben werden. Dieser Prozess wird rekursiv auf beide Teilbereiche angewendet, bis die Liste vollständig sortiert ist. Obwohl Quick Sort für seine Geschwindigkeit bekannt ist, kann seine Instabilität ihn für bestimmte Anwendungen ungeeignet machen.

Obwohl Quick Sort im Durchschnitt recht schnell ist (O(N log N)), weist er, wie die restlichen Algorithmen, im Worst-Case eine Zeitkomplexität von O(N2) auf und ist daher für einen so großen Datensatz nicht geeignet. Sein Speicherbedarf ist jedoch dank der In-Place-Eigenschaft mit einer Speicherkomplexität von O(1) sehr gering.

Merge Sort

Merge Sort ist ein effizienter, stabiler und parallelisierbarer Sortieralgorithmus. Er arbeitet nach dem Prinzip des "Divide and Conquer", bei dem die Liste wiederholt in zwei Hälften geteilt wird, bis jede Teilmenge nur noch ein Element enthält. Anschließend werden die sortierten Teillisten zusammengeführt, wobei die Elemente in der richtigen Reihenfolge angeordnet werden. Die Struktur des Merge Sort Algorithmus ermöglicht eine einfache Parallelisierung, da verschiedene Teile der Liste unabhängig voneinander sortiert werden können, bevor sie wieder zusammengeführt werden. Diese Eigenschaft macht Merge Sort besonders gut geeignet für die Verarbeitung großer Datenmengen in parallelen Umgebungen.

Merge Sort weist eine Zeitkomplexität von O(N log N) auf, was ihn effizient bei großen Datensätzen macht. Der Algorithmus benötigt jedoch zusätzlichen Speicherplatz (O(N)) für die Zwischenspeicherung der Teillisten während des Mergens. Trotzdem bietet Merge Sort den Vorteil einer stabilen Sortierung und einer effizienten Leistung, insbesondere bei großen Datenmengen. Aufgrund der Vorteile und seiner Parallelisierbarkeit werde ich diesen Algorithmus für mein Problem implementieren.

Lösungsimplementierung

Aggregator Klasse

```
public class Aggregator {

    /** Erzeugt ArrayList<Row> mit Rezept-Varianten mit höchster
     * RatePerMinOut1 für jedes Item**/
    public static ArrayList<Row> aggregate(ArrayList<Row> rows) {

        ArrayList<Row> aggregatedRows = new ArrayList<>();
        HashMap<String, ArrayList<Row>> uniqueRecipes = new HashMap<>();

        //Rezepte werden nach Output-Item in HashMap sortiert
        for (Row row : rows) {
            if (!uniqueRecipes.containsKey(row.Recipe)) {
                uniqueRecipes.put(row.Recipe, new ArrayList<>());
            }
            uniqueRecipes.get(row.Recipe).add(row);
        }

        //Aus der HashMap wird für jedes Rezept das beste ausgewählt
        for (Map.Entry<String, ArrayList<Row>> entry: uniqueRecipes.entrySet()) {
            Row highestProdRate = new Row();
            if (entry.getValue().size() > 1){ //Wenn mehrere Rezept-Varianten
                highestProdRate = Aggregator.getHighRate(entry.getValue());
            } else { //Wenn nur ein Rezept
                highestProdRate = entry.getValue().get(0);
            }
            aggregatedRows.add(highestProdRate);
        }
        return aggregatedRows;
    }

    /**Wählt das Rezept mit höchster RatePerMinOut1 aus ArrayList<Row> aus**/
    private static Row getHighRate(ArrayList<Row> rows){
        Row maxRow = new Row();

        for (Row row: rows){
            if (row.RatePerMinOut1 > maxRow.RatePerMinOut1){
                maxRow = row;
            }
        }
        return maxRow;
    }
}
```

Sorter Klasse

```
public class Sorter {
    /** Sortiert die gegebene ArrayList<Row> absteigend nach der
    RatePerMinOut1 eines jeden Rezeptes **/
    public static void sort(ArrayList<Row> rows){
        int length = rows.size();
        if (length < 2){ //Abbruchbedingung der Rekursion
            return;
        }
        //Split als Grundlage von Merge Sort
        int middleIndex = length / 2;
        ArrayList<Row> leftSide = new ArrayList<>();
        ArrayList<Row> rightSide = new ArrayList<>();
        for (int i = 0; i < length; i++){
            if (i < middleIndex){
                leftSide.add(rows.get(i));
            } else {
                rightSide.add(rows.get(i));
            }
        }
        //Rekursiver Aufruf für die jeweilige Seite
        sort(leftSide);
        sort(rightSide);
        merge(rows, leftSide, rightSide); //Tatsächliches sortieren
    }
```

```
    /** Sortiert zwei gegeben ArrayList<Row> in ein Output
    ArrayList<Row> nach RatePerMinOut1 **/
    private static void merge(ArrayList<Row> rows,
        ArrayList<Row> leftSide, ArrayList<Row> 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 += 1;
                continue;
            } else if (rightIndex >= rightSide.size()){
                rows.set(i, leftSide.get(leftIndex));
                leftIndex += 1;
                continue;
            } else if (leftSide.get(leftIndex).RatePerMinOut1 >=
                            rightSide.get(rightIndex).RatePerMinOut1){
                rows.set(i, leftSide.get(leftIndex));
                leftIndex += 1;
                continue;
            } else if (leftSide.get(leftIndex).RatePerMinOut1 <=
                            rightSide.get(rightIndex).RatePerMinOut1){
                rows.set(i, rightSide.get(rightIndex));
                rightIndex += 1;
                continue;
            }
        }
    }
}
```

Row Klasse

```
public class Row {
    public String Recipe;
    public double ProductionTime;
    public String Machine;
    public String Output1;
    public double RatePerMinOut1;
    public String Output2;
    public double RatePerMinOut2;
    public String Input1;
    public double RateIn1;
    public String Input2;
    public double RateIn2;
    public String Input3;
    public double RateIn3;
    public String Input4;
    public double RateIn4;
}
```

AggregatedRow Klassse

Da die eigentlichen Daten in den Rows nicht verändert, sondern nur gefiltert und neu sortiert werden, ist die Implementierung einer zusätzlichen AggregatedRow-Klasse nicht notwendig. Stattdessen wird direkt mit den ursprünglichen Row-Objekten gearbeitet, um die Komplexität des Codes zu reduzieren und die Effizienz zu steigern. Diese Vorgehensweise wurde vorab besprochen und als angemessen erachtet.

Schluss

In Anbetracht der Vielzahl von Sortieralgorithmen und ihrer verschiedenen Eigenschaften ist die Auswahl des richtigen Algorithmus entscheidend für die Effizienz eines Systems. Nach sorgfältiger Analyse habe ich mich für den Merge Sort Algorithmus entschieden. Seine niedrige Zeitkomplexität von O(N log N) und die Möglichkeit der parallelen Verarbeitung machen ihn besonders geeignet für mein Projekt. Die Fähigkeit von Merge Sort, große Datensätze effizient zu verarbeiten, war ebenfalls ein ausschlaggebender Faktor. Insgesamt bietet Merge Sort eine ausgewogene Kombination aus Effizienz und Implementierungskomplexität, was ihn zur besten Wahl für die Sortierung meiner Rezeptdatenbank macht.

Die Verwendung von HashMaps im Programmcode hat sich als äußerst nützlich und effektiv erwiesen, da sie eine elegante Möglichkeit zur Speicherung und schnellen Suche meiner Rezepte bieten.

Mit meinen Arbeitsergebnissen bin ich sehr zufrieden und denke, in Zukunft auf diesem Projekt aufbauen zu können, um zusätzliche Funktionalitäten hinzuzufügen.

Für eine ausführlichere Analyse und den Quellcode des Projekts können Sie das GitHub-Repository unter folgendem Link besuchen: https://github.com/ZenKuJa/JavaSatisfactoryAufgabe.git.

Eigenständigkeitserklärung

gemäß Ziffer 1.2.3 der Anlage 1 zu §§ 3, 4 und 5 der Studien- und Prüfungsordnung für die Bachelorstudiengänge im Studienbereich Wirtschaft der Dualen Hochschule Baden-Württemberg vom 29.09.2015.

Ich versichere hiermit, dass ich meinen Bachelorarbeit (bzw. Projektarbeit oder Seminararbeit) mit dem Thema:

Projektbericht-Sortieralgorithmen Satisfactory

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 29.05.2024

_____

Jannes Kurzke