Abschlussprüfung Winter 2024

Mathematisch-Technischer-Softwareentwickler

AUFTRAGSOPTIMIERUNG

Prüfungsbewerber:

Ron Blänkner

Wiesenstraße 5c, 14612 Falkensee

E-Mail: ron.blaenkner@yahoo.de

Tel.: 01573/2129992

Ausbildungsbetrieb:

Dataciders InMediasP

Inhaltsverzeichnis

Übersicht

Problembeschreibung:

Es sollte ein Programm entwickelt werden, welches rechteckige Aufträge auf einer Rolle platziert. Da jede nicht genutzte Fläche als Ausschuss wegfällte, sollte die Verteilung der Aufträge auf einen minimalen Ausschuss hin optimiert werden.
Da die durch die Aufträge verbrauchte Fläche immer die gleiche ist, optimiert das Programm die Verteilung nach der verbrauchten Rollenlänge.

Die Aufträge werden als Textdatei eingelesen, in der außerdem die Rollenbreite und Optimierungstiefe angegeben sind. Die Optimierungstiefe beschränkt die Anzahl von Aufträgen, die gleichzeitig optimiert werden, bevor der Algorithmus mit den nächsten Aufträgen neugestartet wird.

Die Ergebnisse werden in eine Textdatei geschrieben. Es wird außerdem ein gnuplot-Skript erstellt, um das Ergebnis visualisieren zu können.

Problemlösung:

Zur Lösung des Problems wurde der A*-Algorithmus als Grundlage verwendet. Dieser Algorithmus ist bekannt dafür, optimale Lösungen in Suchräumen zu finden, indem er eine Heuristik verwendet, um vielversprechende Pfade priorisiert zu bearbeiten. Im Kontext der Auftragsplatzierung bedeutet dies, dass mögliche Platzierungen der Rechtecke auf der Rolle als Zustände betrachtet werden. Diese Zustände werden in einer PriorityQueue verwaltet, die immer den Zustand mit der niedrigsten geschätzten Kostenfunktion (Heuristik) bevorzugt.

Die Kostenfunktion wird in zwei Komponenten unterteilt:

1. **Bisher erreichte Höhe**: Die Höhe, die bereits durch die platzierte Lösung erreicht wurde.
2. **Bisher verbleibender Ausschuss**: Das Verhältnis zwischen platzierter Fläche und verbrauchter Rollenfläche gibt einen Faktor zwischen 0 und 1.

Das Produkt dieser beiden Komponenten bestimmt die Kosten eines Zustands. Ziel ist es, eine Lösung mit der minimalen Höhe zu finden, die alle Rechtecke unter den gegebenen Bedingungen (Rollenbreite) platziert.

Ablauf des Algorithmus:

1. **Initialisierung**:
    - Für jeden Auftrag wird eine erste Platzierung in beiden möglichen Ausrichtungen (horizontal und vertikal) am Startpunkt (0, 0) vorgenommen. Diese Platzierungen werden als Anfangszustände in die PriorityQueue eingefügt.
2. **Verarbeitung der Zustände**:
    - Der Zustand mit der höchsten Priorität (niedrigsten geschätzten Kosten) wird aus der Queue entnommen.
    - Neue mögliche Zustände (Platzierungen weiterer Aufträge) werden generiert und der Queue hinzugefügt, sofern sie sich nicht mit anderen Aufträgen überschneiden oder die Rolle zur Seite hin verlassen.
3. **Filtern der Lösungsknoten**:
    - Zustände, deren Höhe bereits größer oder gleich der Höhe der besten bekannten Lösung ist, werden ebenfalls verworfen.
4. **Beenden des Algorithmus**:
    - Sobald die PriorityQueue leer ist, wird die beste gefundene Lösung ausgegeben.

Optimierungstiefe:

Um die Laufzeit des Algorithmus bei einer großen Anzahl von Aufträgen zu begrenzen, wird die Optimierung in Blöcken durchgeführt. Die Optimierungstiefe definiert, wie viele Aufträge gleichzeitig optimiert werden. Nach Abschluss eines Blocks wird der Algorithmus mit den nächsten Aufträgen neugestartet, wobei die bereits optimierten Aufträge fixiert bleiben.

Ergebnisse:

Die Ergebnisse des Algorithmus umfassen:

1. Eine Textdatei, die die platzierte Reihenfolge der Aufträge und die verbrauchte Rollenlänge dokumentiert.
2. Ein automatisch generiertes gnuplot-Skript, das die Platzierung der Rechtecke auf der Rolle visualisiert. Dies ermöglicht eine schnelle Überprüfung der gefundenen Lösung und ihrer Effizienz.

Ein Programmablaufplan für den Algorithmus befindet sich im Anhang.

Programmstruktur

Programmarchitektur:

Das Programm ist nach dem EVA-Prinzip aufgebaut, da das Ziel des Programms eine Transformation der Eingabedaten in die gewünschten Ausgabedaten ist.

Eingabe:

Klassen: Eingabe, Aufgabe, Auftrag

Die Eingabeklasse erwartet einen relativen oder absoluten Dateipfad zu einer Textdatei, in der die Aufgabe in folgendem Format beschrieben ist:

// <Bearbeitungstitel>
<Rollenbreite> <Optimierungstiefe>
<Breite>, <Höhe>, <ID>, <Beschreibung>
<Breite>, <Höhe>, <ID>, <Beschreibung>
…

Z.B.:

// Auftrag IHK1
900 6
100, 200, 1, Auftrag A
297, 420, 2, Auftrag B

Die einzelnen Aufträge werden in Records gepackt, welche dann zusammen mit der Rollenbreite und Optimierungstiefe als Aufgaben-Record zurückgegeben werden.

Verarbeitung:

Klassen: Optimierung, LoesungsKnoten, Zustand, Position, Ausrichtung, Punkt

Die Verarbeitung beginnt in der Optimierungsklasse.
Sie erwartet eine Liste von Aufträgen und die Breite der Rolle, um die optimale Verteilung dieser Aufträge auf der Rolle zu finden.
Der Algorithmus für die Schritte wie oben beschrieben aus. Dazu benutzt er die anderen Klasse der Verarbeitung, die das Modell beschreiben.

Ein LoesungsKnoten stellt einen Schritt in Richtung einer Lösung dar. In ihm wird der derzeitige Zustand und die aktuell erreichte Höhe gespeichert. Er berechnet seine eigene Kostenfunktion und stellt weitere auf ihm basierende Knoten zur Verfügung.

Ein Zustand wird definiert durch eine Ansammlung von Positionen. Er kann sich selbst neue Positionen hinzufügen und kontrolliert diese auf Gültigkeit.

Eine Position enthält den platzierten Auftrag, dessen Ankerpunkt und Ausrichtung und berechnet basierend darauf Andockpunkte für weitere Aufträge. Die Position abstrahiert die

Ausrichtung ab, indem sie Funktionen für die Längen der Aufträge in x-, und y-Richtung bereitstellt.

Ausgabe:

Klassen: Ausgabe

Die Ausgabe ist dafür verantwortlich, die Ergebnisse der Optimierung in zwei verschiedenen Formaten zu speichern:

1. **Textdatei**:

   Diese Datei enthält die Details der gefundenen Lösung. Dazu gehören:

- Die benötigte Länge der Rolle
- Die genutzte Fläche in Prozent
- Die Position und Daten der einzelnen Aufträge
- Die verbleibenden Andockpunkte

2. **gnuplot-Skript**:

   Ein Skript, das die Anordnung der Rechtecke auf der Rolle grafisch darstellt. Dieses Skript nutzt die gnuplot-Syntax und kann direkt ausgeführt werden, um eine Visualisierung zu erzeugen.

Die Ausgabe-Klasse abstrahiert die Dateierstellung und gewährleistet, dass die Ergebnisse in einem leicht verständlichen und weiterverwendbaren Format vorliegen.

Main:

Klassen: Main

Die Main-Klasse dient als Einstiegspunkt des Programms. Sie ist eine Art Controller und führt den groben Ablauf gemäß dem EVA-Prinzip durch.

1. Eingabe: Es wird eine Aufgabe eingelesen.
2. Verarbeitung: Diese Aufgabe wird der Optimierung-Klasse übergeben und zu einer Lösung verarbeitet.
3. Ausgabe: Die gefundene Lösung wird gespeichert.

Ein Klassendiagramm des gesamten Programmes befindet sich im Anhang.

Benutzeranleitung

Voraussetzungen:

Java muss installiert sein (Version 17 oder höher)
Es wird gnuplot benötigt, um die gnuplot-Skripte auszuführen.

Vorbereitung

Um das Programm ausführen zu können, benötigt man eine Eingabedatei. Diese muss dem Format auf Seite 4 entsprechen.
Das Programm kommt mit einigen vorgegebenen Beispielen:

aufgabe1.in
aufgabe2.in
aufgabe3.in
test1.in
test2.in
test3.in
test4.in
test5.in
test6.in

Alle Beispiele befinden sich im Projektordner im Ordner input.

Programm ausführen

Im Projektordner befindet sich die Datei Auftragsoptimierung.jar. Diese kann über ein Kommandozeilentool wie Powershell oder CMD benutzt werden.

Drücke Shift + Rechtsklick im Projektordner und wähle in Terminal öffnen.
Gib nun den folgenden Befehl ein:

```
java -jar Auftragsoptimierung.jar <Dateipfad>
```

Der Dateipfad kann dabei relativ oder absolut angegeben werden.

Nachdem das Programm ausgeführt wurde, findet man die gefundene Lösung im Ordner out. Im Ordner txt befinden sich die Textdateien und im Ordner gnu die plt-Dateien.
Mit einem Doppelklick auf eine plt-Datei generiert gnuplot ein Bild von der Lösung zur Veranschaulichung.

Beispielbefehle

```
java -jar Auftragsoptimierung.jar input\test1.in
```

```
java -jar Auftragsoptimierung.jar
"C:\Users\Example\Projects\Auftragsoptimierung\input\aufgabe3.in"
```

Testskript

Im Projektordner befindet sich außerdem die Datei run_all.bat.
Führt man diese mit einem Doppelklick aus, wird ein Terminal geöffnet und das Programm wird mit jeder Datei aus input einmal gestartet.

Dadurch kann man sich alle Ergebnisse der in input enthaltenen Aufgaben generieren lassen. Im Terminal werden für jede Aufgabe die Lösungshöhe und die gebrauchte Zeit angezeigt.

Das Testskript befindet sich auch im Anhang.

Tests

Testbeschreibungen

Test1: Ein Test mit Optimierungstiefe 8, aber auch sehr großer Rollenbreite. Für einen Menschen ist die Lösung sofort offensichtlich – alle Aufträge horizontal hinlegen – aber der Algorithmus braucht durch die hohe Optimierungstiefe sehr lange und stürzt schließlich aufgrund eine OutOfMemoryErrors ab.

Test2: Ein Test mit ausschließlich 100*100 Quadraten. Trotz der relativ hohen Optimierungstiefe von 6 löst der Algorithmus das Problem für 12 Quadrate in unter einer Sekunde.

Test3: Dieser Test enthält ein Rechteck, dessen Höhe und Breite die Rollenbreite überschneiden. Der Nutzer enthält hier entsprechend eine Benachrichtigung und das Programm bricht ab.

Test4-Test10: Diese Testreihe bearbeitet die gleichen Aufträge mit einer Optimierungstiefe von 1-7, um einen Vergleich für die Effektivität und Laufzeit der verschiedenen Optimierungstiefen zu haben.

Aufgabe1-3: Diese Beispiele waren von der IHK vorgegeben. Die Laufzeit des Algorithmus wurde primär an diesen Beispielen getestet.

Alle Testdateien befinden sich ausgeschrieben im Anhang.

Testumgebung

- **Hardware:**
    - Prozessor: AMD Ryzen 7 5800X 8-Core Processor 3.80GHz
    - Arbeitsspeicher (RAM): 32GB – 2*16GB DDR4
    - Festplattenspeicher: Samsung SSD 860 EVO 500GB
    - Betriebssystem: Windows 11 Pro 64-Bit Version 23H2 (Build 22631.4460)
- **Software:**
    - Java-Version: OpenJDK 21.0.5
    - Weitere Abhängigkeiten: gnuplot für die Visualisierung

Testergebnisse

Testname	Anzahl Aufträge	Optimierungstiefe	Ergebnis	Laufzeit
aufgabe1	2	6	29,70 cm	0,05s
aufgabe2	7	8	88,00 cm	42,98s
aufgabe3	10	6	129,40 cm	2,78s
test1	8	8	OutOfMemoryError	-
test2	12	6	40,00 cm	0,21s
test3	7	5	Auftrag passt nicht	-
test4	7	1	130,80 cm	0,05s
test5	7	2	108,80 cm	0,05s
test6	7	3	107,80 cm	0,07s
test7	7	4	98,00 cm	0,09s
test8	7	5	98,00 cm	0,17s
test9	7	6	98,00 cm	1,80s
test10	7	7	85,00 cm	1m 42,511s

Laufzeitabschätzung

Es gibt für jeden Auftrag 2 mögliche Ausrichtungen und mit jedem platzierten Auftrag gibt es einen möglichen Andockpunkt mehr.
Das ergibt $T(n) = n! * \prod_{i=1}^{n} i * 2 = n! * 2^n * n! = (n!)^2 * 2^n$ Möglichkeiten mit n als die Anzahl der Aufträge. Somit beträgt die asymptotische Laufzeit $O((n!)^2)$.

Die Testergebnisse spiegeln dieses Wachstum auch wider. Ab einer Optimierungstiefe von 8 ist die Laufzeit nicht mehr tragbar und sollte verringert werden.

Die tatsächliche Laufzeit kann stark von den Aufträgen abhängen, wie ein Vergleich von aufgabe3 und test2 oder aufgabe2 und test10 zeigen.

Zusammenfassung

Das Programm arbeitet wie erwartet und optimiert die Platzierung rechteckiger Aufträge auf einer Rolle, sodass die Rollenlänge (und damit der Materialverbrauch) minimiert wird. Der A*-Algorithmus hat sich nur bedingt als geeignet erwiesen, um die Lösungsschritte effizient zu priorisieren und zu finden.

Die klassischen Optimierungen des A*-Algorithmus wie z.B. das Sammeln von bereits bekannten Knoten, um bereits erreichte Zustände nicht mehrfach zu untersuchen, haben sich in ihrer Implementierung als zeitaufwändiger herausgestellt als eine Lösung ohne diese.

Durch die schiere Anzahl an Möglichkeiten, Rechtecke zu legen, war es deutlich schneller einfach alle Möglichkeiten durchzugehen und sie weniger zu filtern.

Verbesserungsmöglichkeiten

Trotz der erfolgreichen Lösung des Problems gab es einige Stellen, an denen Optimierungen möglich sind, um die Effizienz weiter zu steigern:

1. **Vermeidung der doppelten Knotenprüfung:**

   Eine der Hauptursachen für die ineffiziente Laufzeit war die Speicherung und Prüfung von bereits bekannten Knoten. In vielen Fällen führte das ständige Überprüfen und Filtern von doppelten Zuständen zu einem größeren Overhead als die tatsächliche Berechnung der nächsten möglichen Knoten. Es könnte sinnvoll sein, diesen Mechanismus zu optimieren, indem ein schnellerer Datenstrukturtyp (z.B. HashSet) verwendet wird, der die Überprüfung auf Duplikate schneller ermöglicht. Zusätzlich könnte eine verbesserte Kostenfunktion dazu führen, die Anzahl der untersuchten Knoten kleiner zu halten.

2. **Kostenfunktion-Verbesserung:**

   Der A*-Algorithmus nutzt normalerweise eine Kostenfunktion, die die "Kosten" für die Erreichung eines Ziels abschätzt. Es gäbe noch einige Faktoren, die man in diese Berechnung noch einfließen lassen könnte, wie z.B. die Anzahl der bisher verwendeten Aufträge, die noch zu verbrauchende Fläche durch Restaufträge oder die abschätzbare zu erwartende minimale Höhe.

3. **Parallelisierung und Multithreading:**

   Der Algorithmus arbeitet derzeit sequentiell, was insbesondere bei der Vielzahl möglicher Knoten und deren Verarbeitungszeit zu Engpässen führen kann. Eine mögliche Verbesserung wäre die Implementierung von Parallelisierung, bei der verschiedene Teile des Lösungsraums gleichzeitig verarbeitet werden. Die Nutzung

von Multithreading oder eines asynchronen Modells könnte hier die Performance erheblich verbessern, indem mehrere Knoten gleichzeitig untersucht werden.

4. **Optimierung der Speicherung von Zuständen und Positionen:**

   Die Speicherung und Verwaltung der Positionen der Aufträge könnte weiter optimiert werden, insbesondere durch Vermeidung redundanter Berechnungen von Andockpunkten oder wiederholten Validierungen. Eine effizientere Datenstruktur für Positionen und Zustände könnte dazu beitragen, die Ausführungszeit zu verkürzen und die Komplexität zu reduzieren.

5. **Klassifizieren von Aufträgen:**

   Für den Fall, dass es mehr Aufträge gibt, als die Optimierungstiefe zulässt, könnte man die Aufträge vorab so klassifizieren und gruppieren, dass sie möglichst gut zusammenpassen.
   Man könnte z.B. Aufträge zusammen gruppieren, die gleiche Höhen/Breiten haben und die zusammen die Rollenbreite gut ausnutzen. Oder wenn ein einzelner Auftrag übrigbleibt, sollte es der sein, am nächsten an die Rollenbreite herankommt. Dadurch würde der Ausschuss weiter minimiert werden.

Fazit

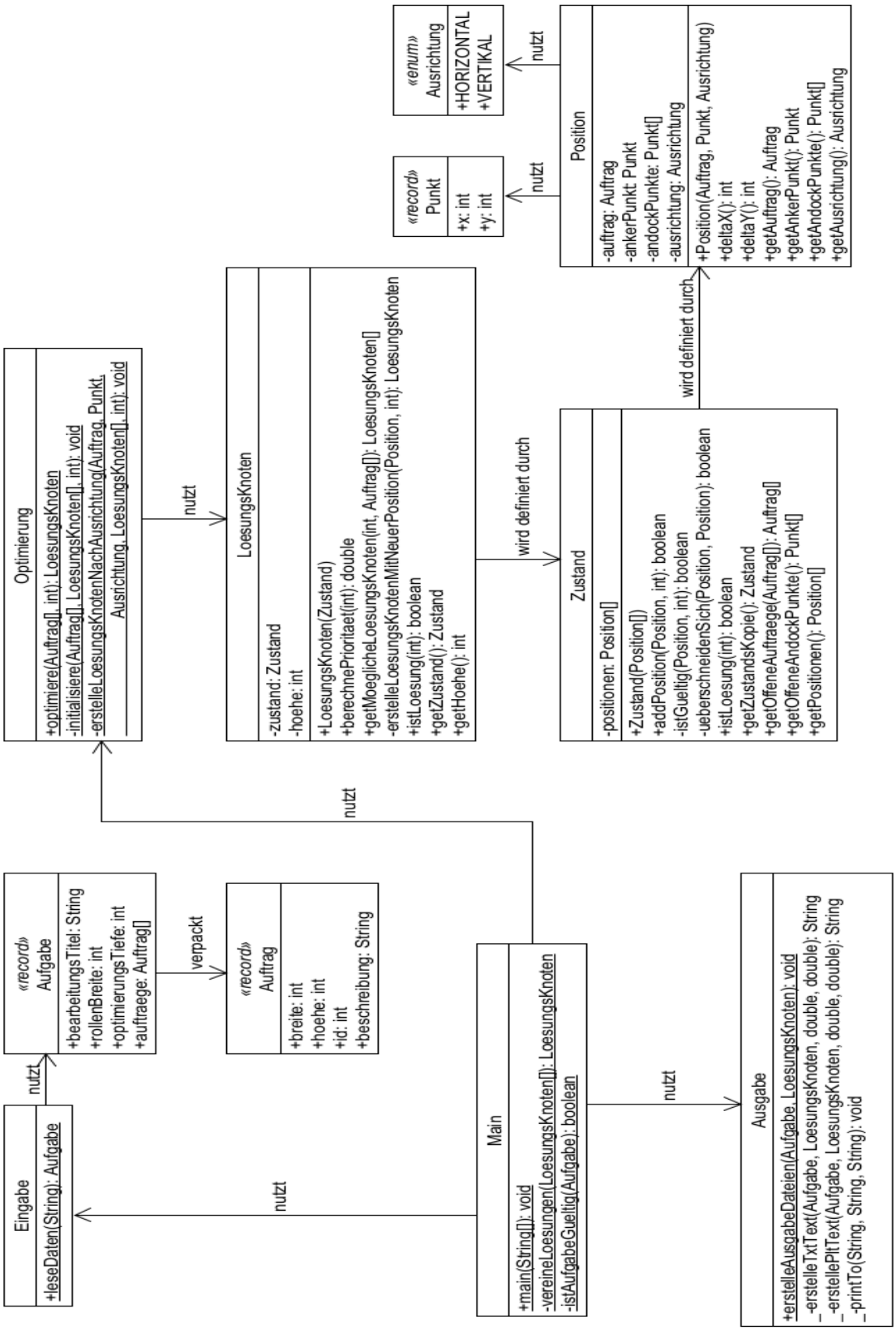
Die Erfahrung hat gezeigt, dass das Sammeln und Filtern bekannter Knoten mehr Zeitaufwand verursachten, als es der tatsächlichen Berechnung der möglichen Lösungsschritte zugutekam. Eine vereinfachte Implementierung, die auf einer breiteren Untersuchung aller möglichen Lösungen basiert, war in diesem Fall effizienter.

Trotzdem gibt es in der Architektur und Implementierung des Programms noch verschiedene Ansatzpunkte für eine weitere Optimierung, wie etwa die Verbesserung der Kostenfunktion, die Parallelisierung der Berechnungen oder eine effizientere Verwaltung der Zustände und Positionen. Diese Verbesserungen könnten dazu beitragen, die Laufzeit erheblich zu verkürzen und das Programm für größere Datensätze oder komplexere Szenarien besser skalierbar zu machen.

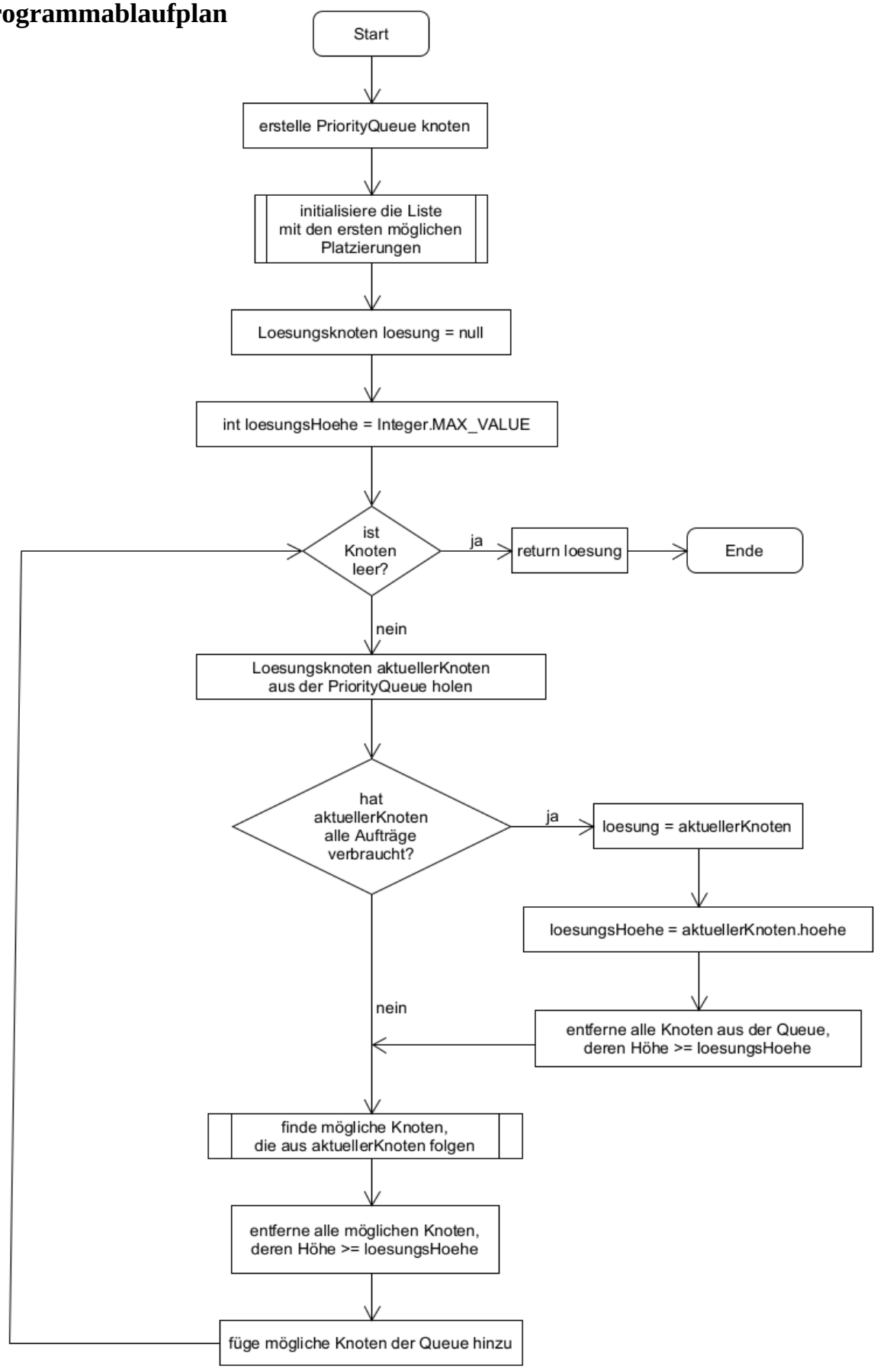
Insgesamt zeigt das Projekt, wie durch den gezielten Einsatz von Algorithmen und Datenstrukturen ein reales Optimierungsproblem gelöst werden kann, wobei jedoch in praktischen Anwendungen oft noch zusätzliche Anpassungen und Optimierungen erforderlich sind, um die besten Ergebnisse zu erzielen.

Anhang

Klassendiagramm



Programmablaufplan

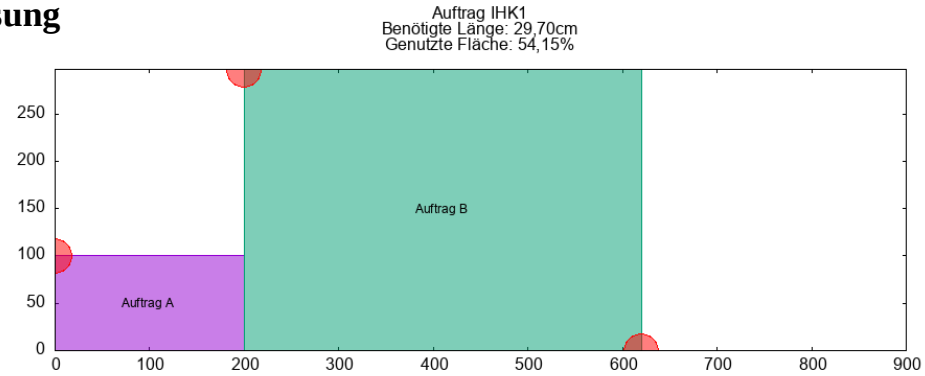


Skript zur Testdurchführung

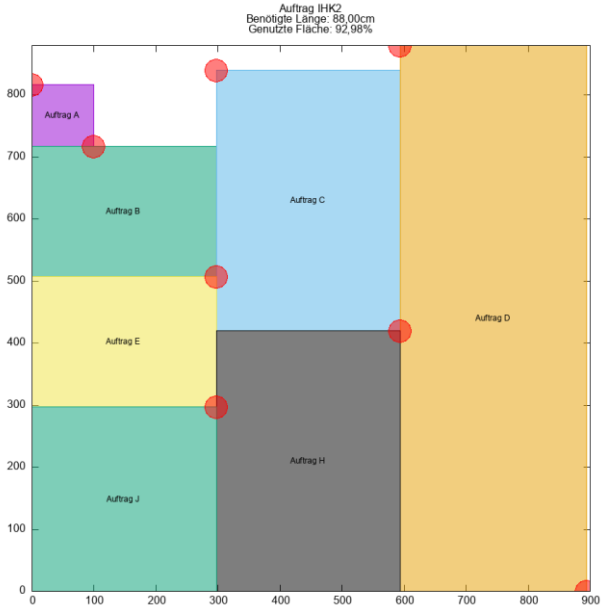
```
@echo off
setlocal enabledelayedexpansion

set "inputFolder=input"
set "jarFile=Auftragsoptimierung.jar"

if not exist "%inputFolder%" (
   echo Der Ordner "%inputFolder%" existiert nicht.
   exit /b 1
)

if not exist "%jarFile%" (
   echo Die Datei "%jarFile%" existiert nicht.
   exit /b 1
)

for %%f in ("%inputFolder%\*") do (
   echo Verarbeite Datei: %%f
   java -jar %jarFile% %%f
)

echo Alle Dateien wurden verarbeitet.
pause
```

Eingabedateien mit Lösung

// Auftrag IHK1
900 6
100, 200, 1, Auftrag A
297, 420, 2, Auftrag B

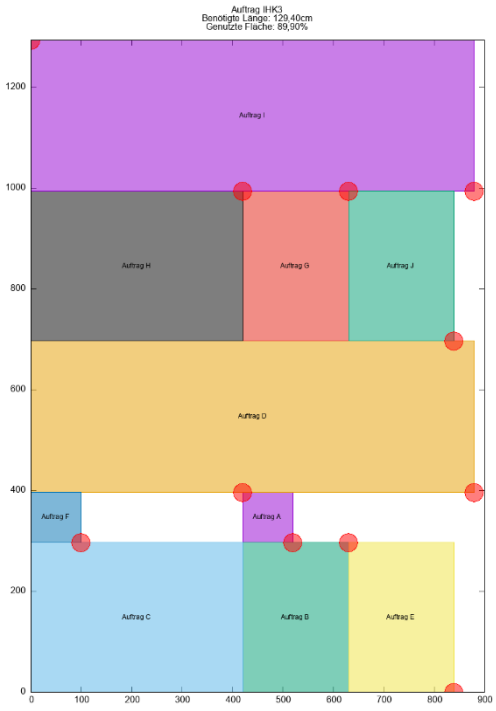


// Auftrag IHK2
900 8
100, 100, 1, Auftrag A
210, 297, 2, Auftrag B
297, 420, 3, Auftrag C
880, 300, 4, Auftrag D
210, 297, 5, Auftrag E
297, 420, 8, Auftrag H
297, 297, 10, Auftrag J



// Auftrag IHK3
900 6
100, 100, 1, Auftrag A

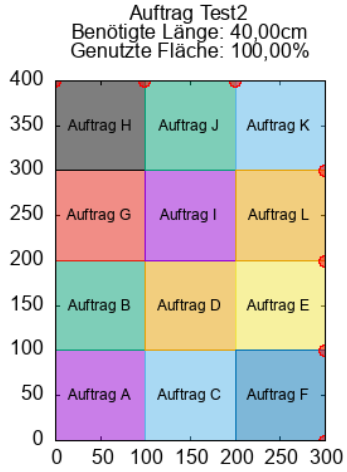
210, 297, 2, Auftrag B
297, 420, 3, Auftrag C
880, 300, 4, Auftrag D
210, 297, 5, Auftrag E
100, 100, 6, Auftrag F
210, 297, 7, Auftrag G
297, 420, 8, Auftrag H
880, 300, 9, Auftrag I
210, 297, 10, Auftrag J



// Auftrag Test1
10000 8
50, 100, 1, Auftrag A
60, 110, 2, Auftrag B
40, 40, 3, Auftrag C
90, 100, 4, Auftrag D
20, 40, 5, Auftrag E
100, 140, 6, Auftrag F
70, 250, 7, Auftrag G
140, 140, 8, Auftrag H -> OutOfMemoryError

// Auftrag Test2

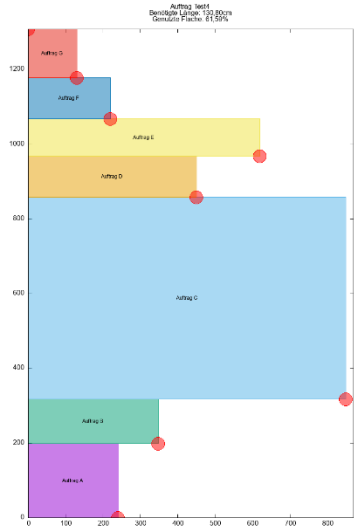
300 6
100, 100, 1, Auftrag A
100, 100, 2, Auftrag B
100, 100, 3, Auftrag C
100, 100, 4, Auftrag D
100, 100, 5, Auftrag E
100, 100, 6, Auftrag F
100, 100, 7, Auftrag G
100, 100, 8, Auftrag H
100, 100, 9, Auftrag I
100, 100, 10, Auftrag J
100, 100, 11, Auftrag K
100, 100, 12, Auftrag L



// Auftrag Test3
800 5
320, 100, 1, Auftrag A
500, 650, 2, Auftrag B
120, 100, 3, Auftrag C
257, 239, 4, Auftrag D
540, 765, 5, Auftrag E
820, 820, 6, Auftrag F
278, 328, 7, Auftrag G -> Auftrag F passt nicht auf die Rolle

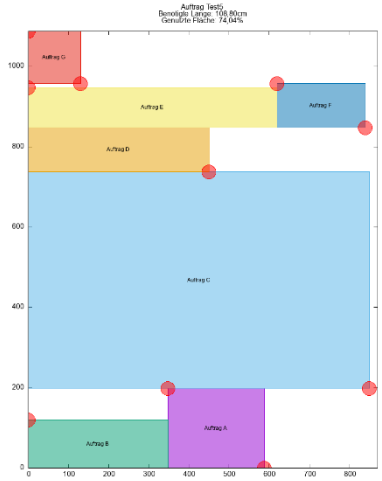
// Auftrag Test4
870 1
240, 198, 1, Auftrag A
348, 120, 2, Auftrag B
850, 540, 3, Auftrag C
450, 110, 4, Auftrag D
620, 100, 5, Auftrag E

220, 110, 6, Auftrag F
130, 130, 7, Auftrag G



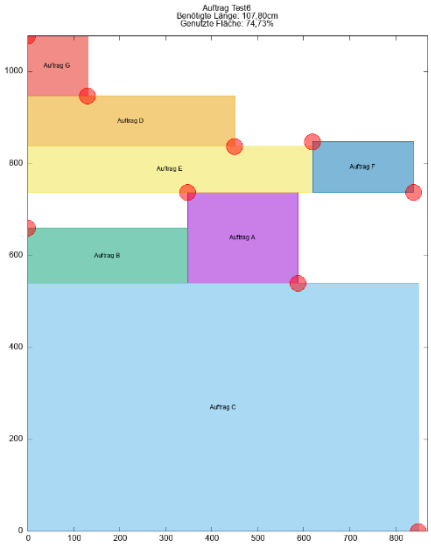
// Auftrag Test5
870 2
240, 198, 1, Auftrag A
348, 120, 2, Auftrag B

850, 540, 3, Auftrag C
450, 110, 4, Auftrag D
620, 100, 5, Auftrag E
220, 110, 6, Auftrag F
130, 130, 7, Auftrag G

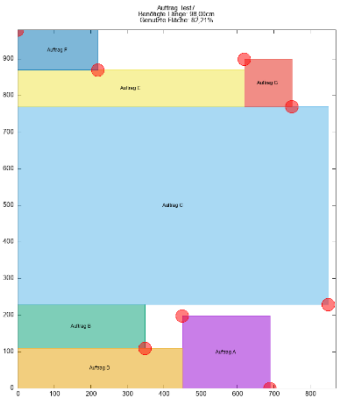


// Auftrag Test6
870 3
240, 198, 1, Auftrag A
348, 120, 2, Auftrag B

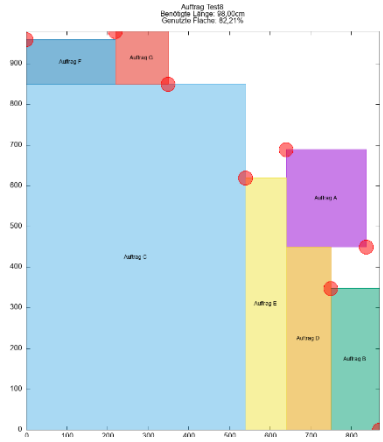
850, 540, 3, Auftrag C
450, 110, 4, Auftrag D
620, 100, 5, Auftrag E
220, 110, 6, Auftrag F
130, 130, 7, Auftrag G



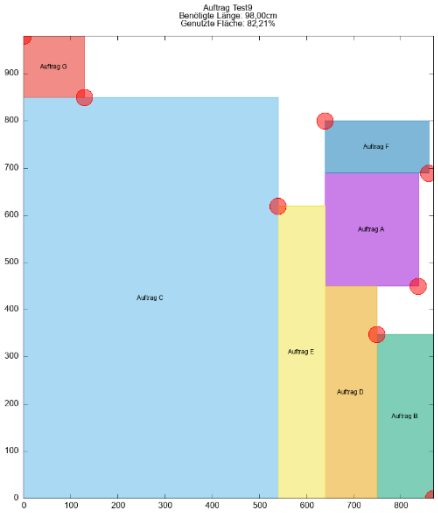
// Auftrag Test7
870 4
240, 198, 1, Auftrag A
348, 120, 2, Auftrag B
850, 540, 3, Auftrag C
450, 110, 4, Auftrag D
620, 100, 5, Auftrag E
220, 110, 6, Auftrag F
130, 130, 7, Auftrag G



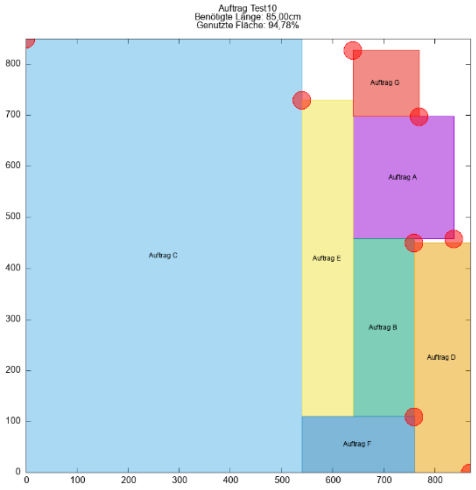
// Auftrag Test8
870 5
240, 198, 1, Auftrag A
348, 120, 2, Auftrag B
850, 540, 3, Auftrag C
450, 110, 4, Auftrag D
620, 100, 5, Auftrag E
220, 110, 6, Auftrag F
130, 130, 7, Auftrag G



// Auftrag Test9
870 6
240, 198, 1, Auftrag A
348, 120, 2, Auftrag B
850, 540, 3, Auftrag C
450, 110, 4, Auftrag D
620, 100, 5, Auftrag E
220, 110, 6, Auftrag F
130, 130, 7, Auftrag G



// Auftrag Test10
870 7
240, 198, 1, Auftrag A
348, 120, 2, Auftrag B
850, 540, 3, Auftrag C
450, 110, 4, Auftrag D
620, 100, 5, Auftrag E
220, 110, 6, Auftrag F
130, 130, 7, Auftrag G



Quellcode (Java)

Main

```
import ausgabe.Ausgabe;
import eingabe.Aufgabe;
import eingabe.Auftrag;
import eingabe.Eingabe;
import verarbeitung.Optimierung;
import verarbeitung.modell.LoesungsKnoten;
import verarbeitung.modell.Position;
import verarbeitung.modell.Punkt;
import verarbeitung.modell.Zustand;

import java.util.ArrayList;
import java.util.List;

public class Main {

  /**
   * Das Programm liest eine Textdatei mit Daten zu rechteckigen Aufträgen ein und optimiert deren Verteilung auf
   * einer Fläche mit fester Breite und variabler Länge so, dass die Länge und die dadurch nicht verbrauchte Fläche
   * minimal ist.
   * <p>
   * Sollte die Anzahl der Aufträge die angegebene Optimierungstiefe überschreiten, werden die Aufträge in Paketen
   * der Größe der Optimierungstiefe aufgeteilt und dem Algorithmus getrennt übergeben.
   * <p>
   * Die daraus resultierenden Ergebnisse werden zum Schluss wieder zusammengefügt, bevor jeweils eine Textdatei und
   * ein Gnuplot-Skript im Ordner /io/out erstellt wird.
   *
   * @param args erwartet einen absoluten oder relativen Pfad zur Eingabedatei
   */
  public static void main(String[] args) {
    if (args.length != 1) {
      System.out.println("Ungültige Parameter. Bitte verwenden Sie den Befehl java -jar Auftragsoptimierung.jar <Dateipfad>");
    }

    long startZeit = System.currentTimeMillis();

    String dateiPfad = args[0];
    List<LoesungsKnoten> loesungen = new ArrayList<>();
    Aufgabe aufgabe = Eingabe.leseDaten(dateiPfad);
```

```
    if (!istAufgabeGueltig(aufgabe)) {
      System.out.println("Ein oder mehr Aufträge passen nicht auf die Rolle!");
      return;
    }

    System.out.println("Starte Optimierung mit Optimierungstiefe: " + aufgabe.optimierungsTiefe());

    while (!aufgabe.auftraege().isEmpty()) {
      List<Auftrag> auftraege = new ArrayList<>();

      for (int i = 0; i < aufgabe.optimierungsTiefe(); i++) {
        if (!aufgabe.auftraege().isEmpty()) {
          auftraege.add(aufgabe.auftraege().remove(0));
        }
      }

      LoesungsKnoten loesung = Optimierung.optimiere(auftraege, aufgabe.rollenBreite());
      System.out.println("Beste Lösungshöhe: " + loesung.getHoehe() + "mm\n");
      loesungen.add(loesung);
    }

    LoesungsKnoten loesung = vereineLoesungen(loesungen);

    Ausgabe.erstelleAusgabeDateien(aufgabe, loesung);

    long endZeit = System.currentTimeMillis();
    double endZeitInSekunden = (endZeit - startZeit) / 1000.0;
    int endZeitInMinuten = 0;
    while (endZeitInSekunden >= 60) {
      endZeitInMinuten++;
      endZeitInSekunden -= 60;
    }
      System.out.printf("\nAusführungszeit: %d Minuten und %.3f Sekunden%n%n",
endZeitInMinuten, endZeitInSekunden);
  }

  /**
   * Überprüft, ob alle Aufträge auf die Rolle passen.
   *
   * @param aufgabe die betrachtete Aufgabe
   * @return true, wenn alle Aufträge auf die Rolle passen, sonst false
   */
  private static boolean istAufgabeGueltig(Aufgabe aufgabe) {
    return aufgabe.auftraege().stream()
        .noneMatch(auftrag -> auftrag.breite() > aufgabe.rollenBreite() && auftrag.hoehe() >
aufgabe.rollenBreite());
  }
```

```
  /**
   * Da die Lösungen alle von einem Startpunkt (0, 0) ausgehen, müssen sie für das Zusammenfügen jeweils auf die
   * maximale Höhe ihres Vorgängers gesetzt werden.
   *
   * @param loesungen eine Liste von optimierten Lösungen
   * @return eine kombinierte Lösung
   */
  private static LoesungsKnoten vereineLoesungen(List<LoesungsKnoten> loesungen) {
    List<Position> kombiniertePositionen = new ArrayList<>();
    int aktuelleHoehe = 0;

    for (LoesungsKnoten loesung : loesungen) {
      for (Position position : loesung.getZustand().getPositionen()) {
        Position verschobenePosition = new Position(
            position.getAuftrag(),
            new Punkt(
                position.getAnkerPunkt().x(),
                position.getAnkerPunkt().y() + aktuelleHoehe
            ),
            position.getAusrichtung()
        );
        kombiniertePositionen.add(verschobenePosition);
      }

      aktuelleHoehe += loesung.getHoehe();
    }

    Zustand kombinierterZustand = new Zustand(kombiniertePositionen);
    return new LoesungsKnoten(kombinierterZustand);
  }
}
```

Eingabe

```
package eingabe;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class Eingabe {
  /**
   * Liest die Daten aus der angegebenen Datei und speichert die Daten in einem Aufgabenrecord.
   *
```

```
    * @param dateiPfad der Name der einzulesenden Datei
    * @return ein Aufgabenrecord mit den ausgelesenen Daten.
    */
   public static Aufgabe leseDaten(String dateiPfad) {
      String bearbeitungTitel;
      int rollenBreite;
      int optimierungsTiefe;
      List<Auftrag> auftraege = new ArrayList<>();

      try (BufferedReader bufferedReader = new BufferedReader(
            new FileReader(dateiPfad))) {

         String line;

         line = bufferedReader.readLine();
         bearbeitungTitel = line.substring(3);

         line = bufferedReader.readLine();
         String[] lineParts = line.split(" ");

         rollenBreite = Integer.parseInt(lineParts[0]);
         optimierungsTiefe = Integer.parseInt(lineParts[1]);

         while ((line = bufferedReader.readLine()) != null) {
            lineParts = line.split(", ");
            int breite = Integer.parseInt(lineParts[0]);
            int hoehe = Integer.parseInt(lineParts[1]);
            int id = Integer.parseInt(lineParts[2]);
            String beschreibung = lineParts[3];

            auftraege.add(new Auftrag(breite, hoehe, id, beschreibung));
         }

         return new Aufgabe(bearbeitungTitel, rollenBreite, optimierungsTiefe, auftraege);

      } catch (IOException e) {
         System.err.println("Die Eingabedatei hat nicht das richtige Format.");
         throw new RuntimeException();
      }
   }
}
```

Aufgabe

```
package eingabe;

import java.util.List;

/**
 * Stellt die Aufgabe dar, die gelöst werden soll.
 *
 * @param bearbeitungTitel  der Titel der Aufgabe
 * @param rollenBreite      die Breite der zu besetzenden Fläche
 * @param optimierungsTiefe die maximale Anzahl an Aufträgen für den Algorithmus auf einmal
 * @param auftraege         die zu verteilenden Rechtecke
 */
public record Aufgabe(String bearbeitungTitel, int rollenBreite, int optimierungsTiefe, List<Auftrag>
auftraege) {
}
```

Auftrag

```
package eingabe;

/**
 * Stellt einen einzelnen Auftrag dar.
 *
 * @param breite      die Breite des Rechtecks
 * @param hoehe       die Höhe des Rechtecks
 * @param id          die ID des Auftrags
 * @param beschreibung eine Beschreibung des Auftrags
 */
public record Auftrag(int breite, int hoehe, int id, String beschreibung) {
}
```

Optimierung

```
package verarbeitung;

import eingabe.Auftrag;
import verarbeitung.modell.*;

import java.util.*;

import static verarbeitung.modell.Ausrichtung.HORIZONTAL;
import static verarbeitung.modell.Ausrichtung.VERTIKAL;

public class Optimierung {

  /**
   * Findet eine optimale Lösung für das Verteilen von Rechtecken auf der Rolle, sodass die
   * Rollenhöhe minimal wird.
   * <p>
   * Der Algorithmus arbeitet mit einer PriorityQueue, die immer die vielversprechendste Lösung nach vorne packt.
   * Der günstigste Lösungspfad wird hierbei durch eine Kombination der bereits erreichten Höhe und dem Anteil
   * der verbrauchten Fläche ermittelt.
   * Dieser Pfad generiert weitere mögliche Pfade und fügt sie der Queue hinzu.
   * <p>
   * Wenn eine Lösung gefunden wurde, werden alle Lösungsknoten, die diese Höhe bereits erreicht haben, aus der
   * Queue eliminiert. Jede neu gefundene Lösung wird mit der alten abgeglichen, um die beste Lösung zu finden.
   * <p>
   * Wenn die Queue leer ist, wurde die Lösung gefunden.
   *
   * @param auftraege    die rechteckigen Aufträge
   * @param rollenBreite die Breite der Rolle
   * @return ein Lösungsknoten, der die optimale Verteilung der Aufträge enthält
   */
  public static LoesungsKnoten optimiere(List<Auftrag> auftraege, int rollenBreite) {
    Queue<LoesungsKnoten> knoten = new PriorityQueue<>(Comparator.comparing(loesungsKnoten -> loesungsKnoten.berechnePrioritaet(rollenBreite)));

    initialisiere(auftraege, knoten, rollenBreite);

    LoesungsKnoten loesung = null;
    int loesungsHoehe = Integer.MAX_VALUE;

    while (!knoten.isEmpty()) {
      LoesungsKnoten aktuellerKnoten = knoten.poll();
```

```
        if (aktuellerKnoten.istLoesung(auftraege.size())) {
            loesung = aktuellerKnoten;
            loesungsHoehe = aktuellerKnoten.getHoehe();
            knoten.removeIf(loesungsKnoten -> loesungsKnoten.getHoehe() >=
aktuellerKnoten.getHoehe());
            System.out.println("Neue beste Lösung gefunden: " + loesungsHoehe + "mm");
        }

        List<LoesungsKnoten> moeglicheKnoten =
aktuellerKnoten.getMoeglicheLoesungsKnoten(rollenBreite, auftraege);

        for (LoesungsKnoten loesungsKnoten : moeglicheKnoten) {
            if(loesungsKnoten.getHoehe() < loesungsHoehe) {
                knoten.add(loesungsKnoten);
            }
        }
    }

    return loesung;
  }

  /**
   * Befüllt die PriorityQueue mit allen Aufträgen in jeder Lage am Startpunkt (0, 0)
   *
   * @param auftraege    die zu verteilenden Aufträge
   * @param knoten       die PriorityQueue mit den Lösungsknoten
   * @param rollenBreite die Breite der Rolle
   */
  private static void initialisiere(List<Auftrag> auftraege, Queue<LoesungsKnoten> knoten, int
rollenBreite) {
    Punkt startPunkt = new Punkt(0, 0);
    for (Auftrag auftrag : auftraege) {
      erstelleLoesungsKnotenNachAusrichtung(auftrag, startPunkt, HORIZONTAL, knoten,
rollenBreite);
      erstelleLoesungsKnotenNachAusrichtung(auftrag, startPunkt, VERTIKAL, knoten, rollenBreite);
    }
  }

  private static void erstelleLoesungsKnotenNachAusrichtung(Auftrag auftrag, Punkt startPunkt,
Ausrichtung ausrichtung, Queue<LoesungsKnoten> knoten, int rollenBreite) {
    Zustand zustand = new Zustand(new ArrayList<>());
    Position position = new Position(auftrag, startPunkt, ausrichtung);

    if (zustand.addPosition(position, rollenBreite)) {
      knoten.add(new LoesungsKnoten(zustand));
    }
  }
}
```

LoesungsKnoten

```
package verarbeitung.modell;

import eingabe.Auftrag;

import java.util.List;
import java.util.Objects;
import java.util.stream.Stream;

public class LoesungsKnoten {

   private final Zustand zustand;
   private final int hoehe;

   /**
    * Die Höhe wird anhand des höchsten Andockpunkts festgemacht.
    *
    * @param zustand die derzeitige Verteilung der Aufträge auf der Rolle
    */
   public LoesungsKnoten(Zustand zustand) {
      this.zustand = zustand;

      hoehe = zustand.getOffeneAndockPunkte().stream()
            .mapToInt(Punkt::y)
            .max().orElseThrow();
   }

   /**
    * Diese Methode berechnet eine Priorität für den Lösungsknoten.
    * Je niedriger diese Priorität ausfällt, desto näher ist der Knoten an einer guten Lösung dran.
    *
    * @param rollenBreite die Breite der Rolle
    * @return die errechnete Priorität
    */
   public double berechnePrioritaet(int rollenBreite) {
      double flaeche = hoehe * rollenBreite;
      double benutzteFlaeche = zustand.getPositionen().stream()
            .map(Position::getAuftrag)
            .mapToInt(auftrag -> auftrag.breite() * auftrag.hoehe())
            .sum();

      double benutzteFlaecheInProzent = benutzteFlaeche / flaeche;

      return hoehe * (1 - benutzteFlaecheInProzent);
   }
```

```
  /**
   * Findet vom derzeitigen Zustand aus alle möglichen nächsten Schritte.
   *
   * @param rollenBreite die Breite der Rolle
   * @param auftraege    die Liste der Aufträge
   * @return eine Liste von hierauf folgenden Lösungsknoten
   */
  public List<LoesungsKnoten> getMoeglicheLoesungsKnoten(int rollenBreite, List<Auftrag>
auftraege) {
     List<Auftrag> offeneAuftraege = zustand.getOffeneAuftraege(auftraege);
     List<Punkt> offeneAndockPunkte = zustand.getOffeneAndockPunkte();

     List<Position> neuePositionen = offeneAuftraege.stream()
           .flatMap(auftrag -> offeneAndockPunkte.stream()
                 .flatMap(andockPunkt -> Stream.of(Ausrichtung.values())
                       .map(ausrichtung -> new Position(auftrag, andockPunkt, ausrichtung))))
           .toList();

     return neuePositionen.stream()
           .map(position -> erstelleLoesungsKnotenMitNeuerPosition(position, rollenBreite))
           .filter(Objects::nonNull)
           .toList();
  }

  /**
   * Erstellt aus dem derzeitigen Zustand den nächsten Lösungsknoten mit der übergebenden
Position.
   * Wenn die neue Position einen ungültigen Zustand ergibt, wird null zurückgegeben.
   *
   * @param neuePosition die nächste Position
   * @param rollenBreite die Breite der Rolle
   * @return einen hierauf folgenden Lösungsknoten, wenn möglich, sonst null
   */
  private LoesungsKnoten erstelleLoesungsKnotenMitNeuerPosition(Position neuePosition, int
rollenBreite) {
     Zustand zustandsKopie = zustand.getZustandsKopie();
     if (zustandsKopie.addPosition(neuePosition, rollenBreite)) {
        return new LoesungsKnoten(zustandsKopie);
     }
     return null;
  }

  /**
   * Ein Lösungsknoten ist am Ende seines Weges, wenn alle Aufträge verwendet wurden.
   *
   * @param anzahlAuftraege die Anzahl der verwendeten Aufträge
   * @return true, wenn alle Aufträge verwendet wurden, sonst false
```

```
    */
   public boolean istLoesung(int anzahlAuftraege) {
      return zustand.istLoesung(anzahlAuftraege);
   }

   public Zustand getZustand() {
      return zustand;
   }

   public int getHoehe() {
      return hoehe;
   }
}
```

Zustand

```
package verarbeitung.modell;

import eingabe.Auftrag;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;

public class Zustand {

   private final List<Position> positionen;

   public Zustand(List<Position> positionen) {
      this.positionen = positionen;
   }

   /**
    * Fügt dem Zustand eine neue Position hinzu. Es wird überprüft, ob dies einen gültigen Zustand
ergibt, bevor
    * sie eingefügt wird.
    *
    * @param neuePosition die einzufügende Position
    * @param rollenBreite die Breite der Rolle für die Überprüfung
    * @return true, wenn die Position angenommen wurde, sonst false
    */
   public boolean addPosition(Position neuePosition, int rollenBreite) {
      if (istGueltig(neuePosition, rollenBreite)) {
         positionen.add(neuePosition);
         positionen.sort(Comparator.comparingInt(position -> position.getAuftrag().id()));
         return true;
      }
```

```
        return false;
    }

    /**
     * Überprüft, ob eine neue Position innerhalb der Rolle liegt und sich mit keiner anderen Position
     * überschneidet.
     *
     * @param neuePosition die neu einzufügende Position
     * @param rollenBreite die Breite der Rolle
     * @return true, wenn beide Bedingungen zutreffen, sonst false
     */
    private boolean istGueltig(Position neuePosition, int rollenBreite) {
        boolean innerhalbRollenBreite = neuePosition.getAndockPunkte().stream()
                .mapToInt(Punkt::x)
                .noneMatch(xWert -> xWert > rollenBreite);

        boolean keineUeberschneidungen = positionen.stream()
                .noneMatch(position -> ueberschneidenSich(position, neuePosition));

        return innerhalbRollenBreite && keineUeberschneidungen;
    }

    /**
     * Überprüft, ob sich zwei Positionen überschneiden, indem ihre Grenzen links, rechts, oben und
unten
     * miteinander verglichen werden.
     *
     * @param p1 die erste Position
     * @param p2 die zweite Position
     * @return true, wenn sich die Positionen überschneiden, sonst false
     */
    private boolean ueberschneidenSich(Position p1, Position p2) {
        int links1 = p1.getAnkerPunkt().x();
        int rechts1 = links1 + p1.deltaX();
        int unten1 = p1.getAnkerPunkt().y();
        int oben1 = unten1 + p1.deltaY();

        int links2 = p2.getAnkerPunkt().x();
        int rechts2 = links2 + p2.deltaX();
        int unten2 = p2.getAnkerPunkt().y();
        int oben2 = unten2 + p2.deltaY();

        boolean xUeberlappt = rechts1 > links2 && rechts2 > links1;
        boolean yUeberlappt = oben1 > unten2 && oben2 > unten1;

        return xUeberlappt && yUeberlappt;
    }
```

```
  /**
   * Der Zustand ist eine Lösung, wenn alle Aufträge verbraucht wurden.
   *
   * @param anzahlAuftraege die Gesamtzahl der Aufträge
   * @return true, wenn es so viele Positionen wie Aufträge gibt, sonst false
   */
  public boolean istLoesung(int anzahlAuftraege) {
     return positionen.size() == anzahlAuftraege;
  }

  /**
   * Erstellt eine Kopie des Zustandes, zur weiteren Verwendung in einem folgenden Lösungsknoten.
   *
   * @return die Kopie des Zustandes
   */
  public Zustand getZustandsKopie() {
     return new Zustand(new ArrayList<>(positionen));
  }

  /**
   * Filtert die Liste der Aufträge nach Aufträgen, die noch nicht verwendet wurden.
   *
   * @param auftraege die Liste der Aufträge
   * @return eine Liste mit nicht verwendeten Aufträgen
   */
  public List<Auftrag> getOffeneAuftraege(List<Auftrag> auftraege) {
     List<Auftrag> benutzteAuftraege = positionen.stream()
           .map(Position::getAuftrag)
           .toList();

     return auftraege.stream()
           .filter(auftrag -> !benutzteAuftraege.contains(auftrag))
           .toList();
  }

  /**
   * Sucht alle noch nicht verwendeten Andockpunkte, indem sie mit den Ankerpunkten verglichen
werden.
   *
   * @return alle noch nicht verwendeten Andockpunkte des Zustands.
   */
  public List<Punkt> getOffeneAndockPunkte() {
     List<Punkt> alleAndockPunkte = positionen.stream()
           .flatMap(position -> position.getAndockPunkte().stream())
           .distinct()
           .toList();

     List<Punkt> alleAnkerpunkte = positionen.stream()
```

```
            .map(Position::getAnkerPunkt)
            .toList();

        return alleAndockPunkte.stream()
            .filter(andockPunkt -> !alleAnkerpunkte.contains(andockPunkt))
            .toList();
    }

    public List<Position> getPositionen() {
        return positionen;
    }
}
```

Position

```
package verarbeitung.modell;

import eingabe.Auftrag;

import java.util.ArrayList;
import java.util.List;

import static verarbeitung.modell.Ausrichtung.VERTIKAL;

public class Position {

    private final Auftrag auftrag;
    private final Punkt ankerPunkt;
    private final List<Punkt> andockPunkte;
    private final Ausrichtung ausrichtung;

    /**
     * Eine Position stellt die Position eines Auftrags dar anhand seines Ankerpunkts und Ausrichtung.
     * Die daraus resultierenden Ankerpunkte werden ebenso gespeichert.
     *
     * @param auftrag     der betreffende Auftrag
     * @param ankerPunkt  der Ankerpunkt des Auftrags
     * @param ausrichtung die Ausrichtung des Auftrags
     */
    public Position(Auftrag auftrag, Punkt ankerPunkt, Ausrichtung ausrichtung) {
        this.auftrag = auftrag;
        this.ankerPunkt = ankerPunkt;
        this.ausrichtung = ausrichtung;

        andockPunkte = new ArrayList<>();
        andockPunkte.add(new Punkt(ankerPunkt.x(), ankerPunkt.y() + deltaY()));
        andockPunkte.add(new Punkt(ankerPunkt.x() + deltaX(), ankerPunkt.y()));
```

```
    }

    /**
     * Liefert in Abhängigkeit seiner Ausrichtung die Länge des Auftrags in x-Richtung.
     *
     * @return die Länge des Auftrags in x-Richtung
     */
    public int deltaX() {
        return ausrichtung == VERTIKAL ? auftrag.breite() : auftrag.hoehe();
    }

    /**
     * Liefert in Abhängigkeit seiner Ausrichtung die Länge des Auftrags in y-Richtung.
     *
     * @return die Länge des Auftrags in y-Richtung
     */
    public int deltaY() {
        return ausrichtung == VERTIKAL ? auftrag.hoehe() : auftrag.breite();
    }

    public Auftrag getAuftrag() {
        return auftrag;
    }

    public Punkt getAnkerPunkt() {
        return ankerPunkt;
    }

    public List<Punkt> getAndockPunkte() {
        return andockPunkte;
    }

    public Ausrichtung getAusrichtung() {
        return ausrichtung;
    }
}
```

Punkt

```
package verarbeitung.modell;

public record Punkt(int x, int y) {
}
```

Ausrichtung

```
package verarbeitung.modell;

/**
 * Ein Enum für die verschiedenen Ausrichtungen der Rechtecke.
 */
public enum Ausrichtung {
   HORIZONTAL,
   VERTIKAL
}
```

Ausgabe

```
package ausgabe;

import eingabe.Aufgabe;
import verarbeitung.modell.LoesungsKnoten;
import verarbeitung.modell.Position;
import verarbeitung.modell.Punkt;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class Ausgabe {

   /**
    * Schreibt die Lösung in eine Textdatei und erstellt ein Skript für gnuplot.
    *
    * @param aufgabe die ursprüngliche Aufgabe
    * @param loesung die zu speichernde Lösung
    */
   public static void erstelleAusgabeDateien(Aufgabe aufgabe, LoesungsKnoten loesung) {
      double laenge = loesung.getHoehe();
      int genutzteFlaeche = loesung.getZustand().getPositionen().stream()
            .map(Position::getAuftrag)
            .mapToInt(auftrag -> auftrag.breite() * auftrag.hoehe())
            .sum();
      int vorhandeneFlaeche = loesung.getHoehe() * aufgabe.rollenBreite();
      double genutzteFlaecheInProzent = (double) genutzteFlaeche * 100 / vorhandeneFlaeche;
      String titel = aufgabe.bearbeitungTitel().toLowerCase().replace(" ", "_");

      String txtText = erstelleTxtText(aufgabe, loesung, laenge, genutzteFlaecheInProzent);
      String pltText = erstellePltText(aufgabe, loesung, laenge, genutzteFlaecheInProzent);
```

```
    printTo("out/txt", titel + ".out", txtText);
    printTo("out/gnu", titel + ".plt", pltText);
  }

  /**
   * Erstellt einen Text nach Vorgabe mit der Lösung.
   *
   * @param aufgabe              die ursprüngliche Aufgabe
   * @param loesung              die gefundene Lösung
   * @param laenge               die Länge der besetzten Fläche
   * @param genutzteFlaecheInProzent die genutzte Fläche in Prozent
   * @return der erstellte Text
   */
  private static String erstelleTxtText(Aufgabe aufgabe, LoesungsKnoten loesung, double laenge,
double genutzteFlaecheInProzent) {
    StringBuilder txtText = new StringBuilder(String.format("""
        // %s
        Benötigte Länge: %.2fcm
        Genutzte Fläche: %.2f%%
        \s
        Positionierung der Kundenaufträge:
        """, aufgabe.bearbeitungTitel(), laenge / 10, genutzteFlaecheInProzent));

    for (Position position : loesung.getZustand().getPositionen()) {
      int linksUntenX = position.getAnkerPunkt().x();
      int linksUntenY = position.getAnkerPunkt().y();
      int rechtsObenX = position.getAnkerPunkt().x() + position.deltaX();
      int rechtsObenY = position.getAnkerPunkt().y() + position.deltaY();

      txtText.append("\t")
          .append(linksUntenX).append(" ")
          .append(linksUntenY).append(" ")
          .append(rechtsObenX).append(" ")
          .append(rechtsObenY).append(" - ")
          .append(position.getAuftrag().id()).append(" - ")
          .append(position.getAuftrag().beschreibung()).append("\n");
    }

    txtText.append("\n\nVerbleibende Andockpunkte:\n");

    for (Punkt punkt : loesung.getZustand().getOffeneAndockPunkte()) {
      txtText.append("\t")
          .append(punkt.x()).append(" ")
          .append(punkt.y()).append("\n");
    }

    return txtText.toString();
  }
```

```
  /**
   * Erstellt ein Skript für gnuplot nach Vorgabe mit der Lösung.
   *
   * @param aufgabe              die ursprüngliche Aufgabe
   * @param loesung              die gefundene Lösung
   * @param laenge               die Länge der besetzten Fläche
   * @param genutzteFlaecheInProzent die genutzte Fläche in Prozent
   * @return das erstellte Skript
   */
  private static String erstellePltText(Aufgabe aufgabe, LoesungsKnoten loesung, double laenge,
double genutzteFlaecheInProzent) {
    StringBuilder pltText = new StringBuilder(String.format("""
            reset
            set term png size %d,%d
            set output '%s'
            set xrange [0:%d]
            set yrange [0:%d]
            set size ratio -1
            \s
            set title "\\
            %s\\n\\
            Benötigte Länge: %.2fcm\\n\\
            Genutzte Fläche: %.2f%%"
            \s
            set style fill transparent solid 0.5 border
            set key noautotitle
            \s
            $data <<EOD
            # x_LU y_LU x_RO y_RO Auftragsbeschreibung ID
            """, aufgabe.rollenBreite(), (int) laenge + 100, aufgabe.bearbeitungTitel() + ".png",
aufgabe.rollenBreite(),
        loesung.getHoehe(), aufgabe.bearbeitungTitel(), laenge / 10, genutzteFlaecheInProzent));

    for (Position position : loesung.getZustand().getPositionen()) {
      int linksUntenX = position.getAnkerPunkt().x();
      int linksUntenY = position.getAnkerPunkt().y();
      int rechtsObenX = position.getAnkerPunkt().x() + position.deltaX();
      int rechtsObenY = position.getAnkerPunkt().y() + position.deltaY();

      pltText.append(linksUntenX).append(" ")
          .append(linksUntenY).append(" ")
          .append(rechtsObenX).append(" ")
          .append(rechtsObenY).append(" \"")
          .append(position.getAuftrag().beschreibung()).append("\" ")
          .append(position.getAuftrag().id()).append("\n");
    }
```

```
        pltText.append("""
                EOD
                \s
                $anchor <<EOD
                # x y
                """);

        for (Punkt punkt : loesung.getZustand().getOffeneAndockPunkte()) {
            pltText.append(punkt.x()).append(" ")
                    .append(punkt.y()).append("\n");
        }

        pltText.append("""
                EOD
                \s
                plot \\
                '$data' using (($3-$1)/2+$1):(($4-$2)/2+$2):(($3-$1)/2):(($4-$2)/2):6 \\
                        with boxxy linecolor var, \\
                '$data' using (($3-$1)/2+$1):(($4-$2)/2+$2):5 \\
                        with labels font "arial,9", \\
                '$anchor' using 1:2 with circles lc rgb "red"
                """);

        return pltText.toString();
    }

    /**
     * Schreibt den übergebenen Text in den angegebenen Pfad.
     *
     * @param pfad  der zu beschreibende Pfad
     * @param titel der Titel der Datei
     * @param text  der zu speichernde Text
     */
    private static void printTo(String pfad, String titel, String text) {
        Path dateiPfad = Paths.get(pfad, titel);
        try {
            Files.createDirectories(dateiPfad.getParent());
            Files.writeString(dateiPfad, text);
            System.out.println("Datei erstellt: " + dateiPfad.toAbsolutePath());
        } catch (IOException e) {
            System.err.println("Fehler beim Speichern der Datei: " + e.getMessage());
        }
    }
}
```