Rucksackproblem

Inhalt

Einführung

Das Rucksackproblem ist ein kombinatorisches Optimierungsproblem. Der Name im englischen ist Knapsack-Problem.

Es gibt einen Rucksack, der ein Gewichtslimit (T) hat, und eine Menge von Gegenständen, die jeweils ein Gewicht (w) und einen Wert (v) haben. Es wird nun die beste Kombination an Gegenständen gesucht, die zusammen das Gewichtslimit einhalten und dabei den höchsten Wert erzielen. Dies ist die einfachste Variante des Knapsack-Problems und wird, als 0/1 Variante bezeichnet.

Es gibt mehrere Varianten des Rucksackproblem. Varianten sind zum Beispiel die Bounded-Variante, bei der der gleich Gegenstand eine bestimmte Anzahl oft mitgenommen werden kann, und die Unbounded-Variante, bei der das Mitnehmen des gleichen Gegenstands unlimitiert ist.

Diese Ausarbeitung bezieht sich ausschließ auf die 0/1-Variante.

Bei der Variante wird für jeden Gegenstand die Entscheidung getroffen ihn mitzunehmen oder diesen zurückzulassen.

Dies lässt sich einem Baumdiagramm darstellen.

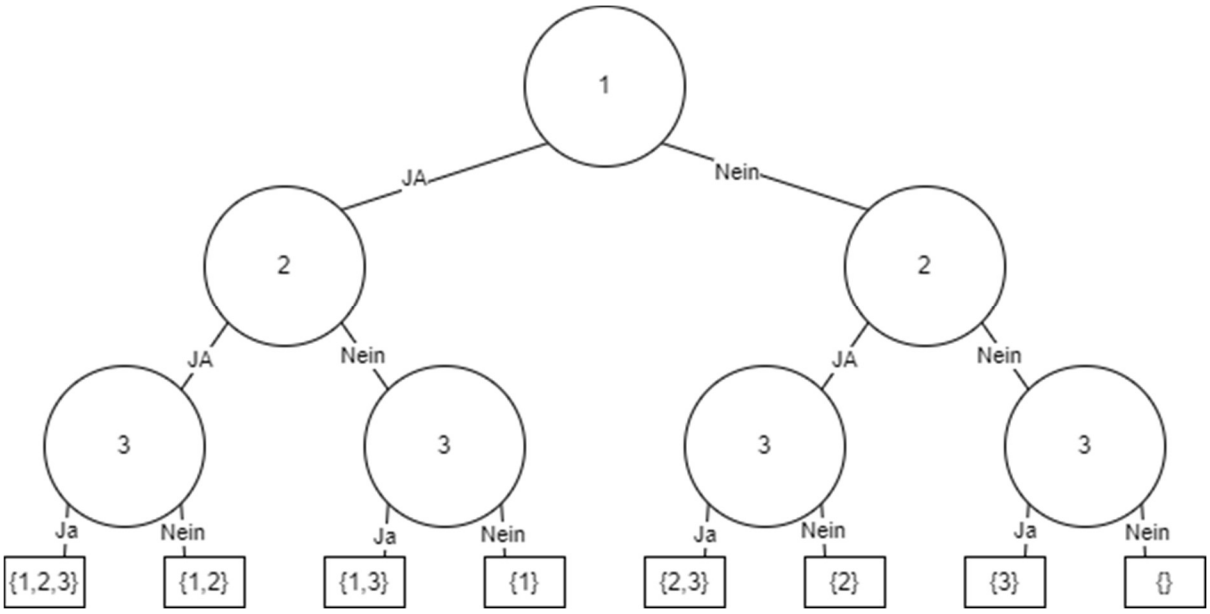


Abbildung 1

Der oben dargestellte Baum wurde für eine Liste von 3 Gegenständen erstellt.

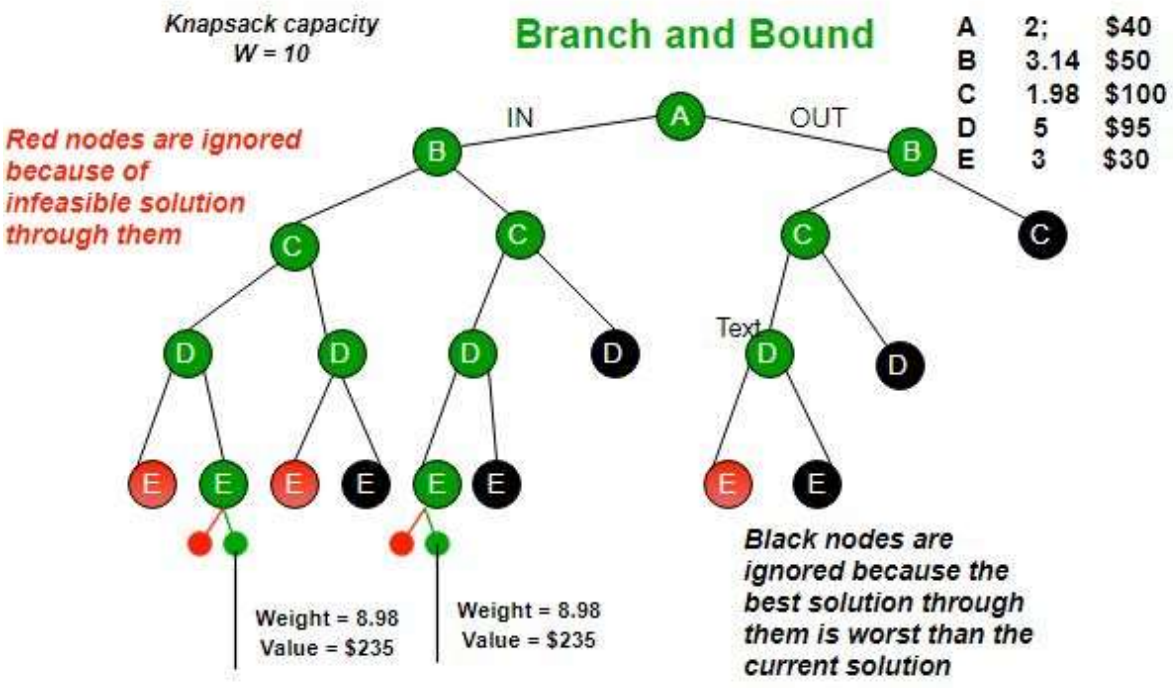
Das 0/1 Knapsack-Problem hat eine Laufzeitkomplexität von 2^n wie in der Abbildung zusehen ist.

Das Problem ist NP-Vollständig, dies bedeutet unmöglich ist ein Ergebnis für das Problem in polynomieller Zeit zu finden.

Standard-Herangehensweisen

Die technisch einfachste Methode ist es alle möglichen Kombinationen zu Generieren und diese alle zu Prüfen. Diese Methode ist erschöpfend und wird Brute-Force genannt. Dieser Ansatz ist jedoch problematisch, da alle Arme des Baumes vollständig durchlaufen werden müssen.

Auch kann dieses Problem mit der Branch and Bound-Methode bearbeitet werden. Dies ist pragmatisch gesehen eine Optimierung der Brute-Force-Methode. Bei diesem Verfahren wird versucht Arme auszuschließen, die keine Möglichkeit haben eine bessere Lösung zu liefern, wie die bisher beste Lösung.



Quelle: https://www.geeksforgeeks.org/branch-and-bound-algorithm/

Die gezeigte Grafik erklärt das Branch and Bounds Verfahren.
Die grünen Kreise zeigen Kombinationen, die vielleicht eine neue beste Kombination generieren können. Rote Kreise sind Gegenstände, bei denen festgestellt wurde, dass alle Kombinationen danach unmöglich sind. In unserem Fall wären die Kombinationen, die immer über dem Gewichtslimit sind. Schwarze Kreise zeigen Arme, die keine Möglichkeit haben eine neue beste Kombination zu generieren. Der beste Wert der Unterkombination plus dem aktuellen Wert des Arms liegen unter dem besten bereits bekannten Wert.

Eine Lösung durch einen Greedy-Algorithmus ist nicht möglich, da die beste Kombination so nicht garantiert werden kann.
Wenn versucht wird, die Lösung mit einem Greedy-Algorithmus zu finden, dann wird es sehr wahrscheinlich passieren, dass nicht die Optimale Lösung gefunden wird.

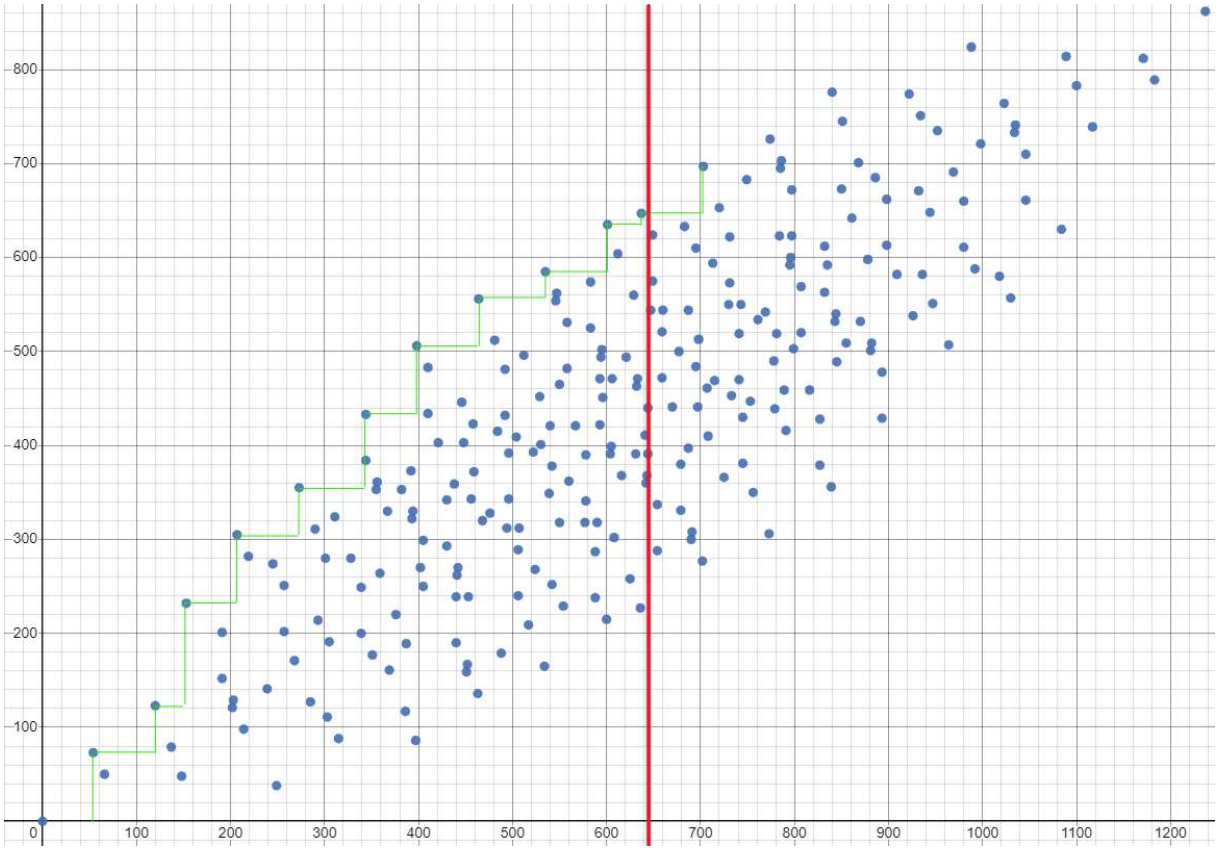
Erklärung Nemhauser-Ullmann Algorithmus

Der Nemhauser-Ullmann Algorithmus löst das Rucksackproblem durch Finden eines Pareto-Optimum.

Ein Pareto-Optimum ist eine Kombination, bei der der Wert am höchsten ist und das Gewicht innerhalb des Gewichtslimit liegt. Eine Kombination ist ein Pareto-Optimum die beiden folgenden Bedingungen erfüllt sind:

$$\sum_{i \in S} w[i] \leq T$$

$$\sum_{i \in S} v[i] \; ist \; maximal$$



Grafik zeigt alle Punkte des späteren Problems (erstellt mit Desmos)

Ein Pareto-Optimum ist ein Punkt, der der beste Wert ist, der in einem bestimmten Bereich erreicht werden kann.

Im Beispiel oben ist eine Auflistung aller Kombinationen, die in der späteren Problem Frage auftreten können. Das Pareto-Optimum sind die Punkte, die mit der hellgrünen Line miteinander verbunden sind. Dies sind die besten Werte, die in einem bestimmten Gewichtsbereich erreicht werden können.

Die Logik des Algorithmus funktioniert wie folgt.

Es gibt eine Liste mit Gegenständen, die nach Gewicht sortiert sein müssen.

Der Nemhauser-Ullmann Algorithmus funktioniert so, dass zwei Listen nebeneinander geführt werden. Die Listen sind L und L‘, wobei L‘ am Anfang eine Kopie von L ist.

Es wird nun versucht nach einem Gegenstand in die Liste einzufügen, wenn dies vom Gewicht her möglich ist. Am Ende werden L‘ und L zusammengefügt und jede Kombination, deren Wert kleiner (dominiert) wird, entfernt.

Dieser Prozess wird nun für alle Gegenstände wiederholt.

Eine genaue Erklärung des genutzten Algorithmus wird später in der „Implementierung und Java Code“-Sektion vorgenommen.

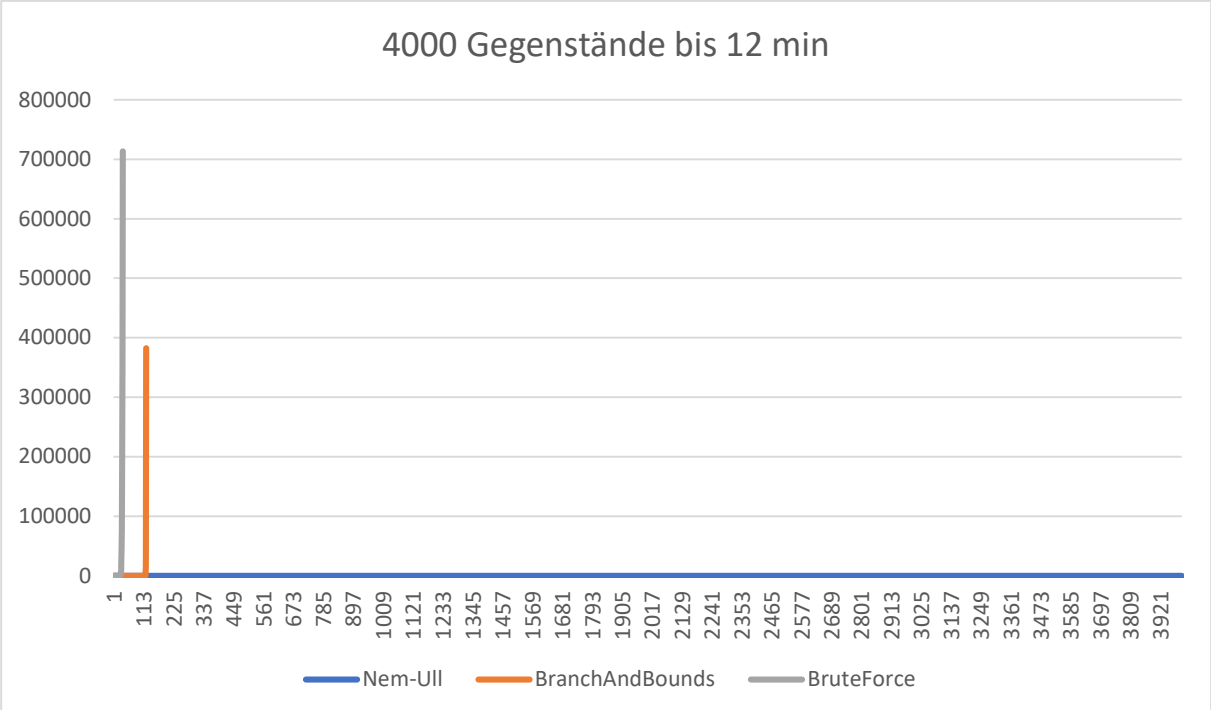
Problembeschreibung

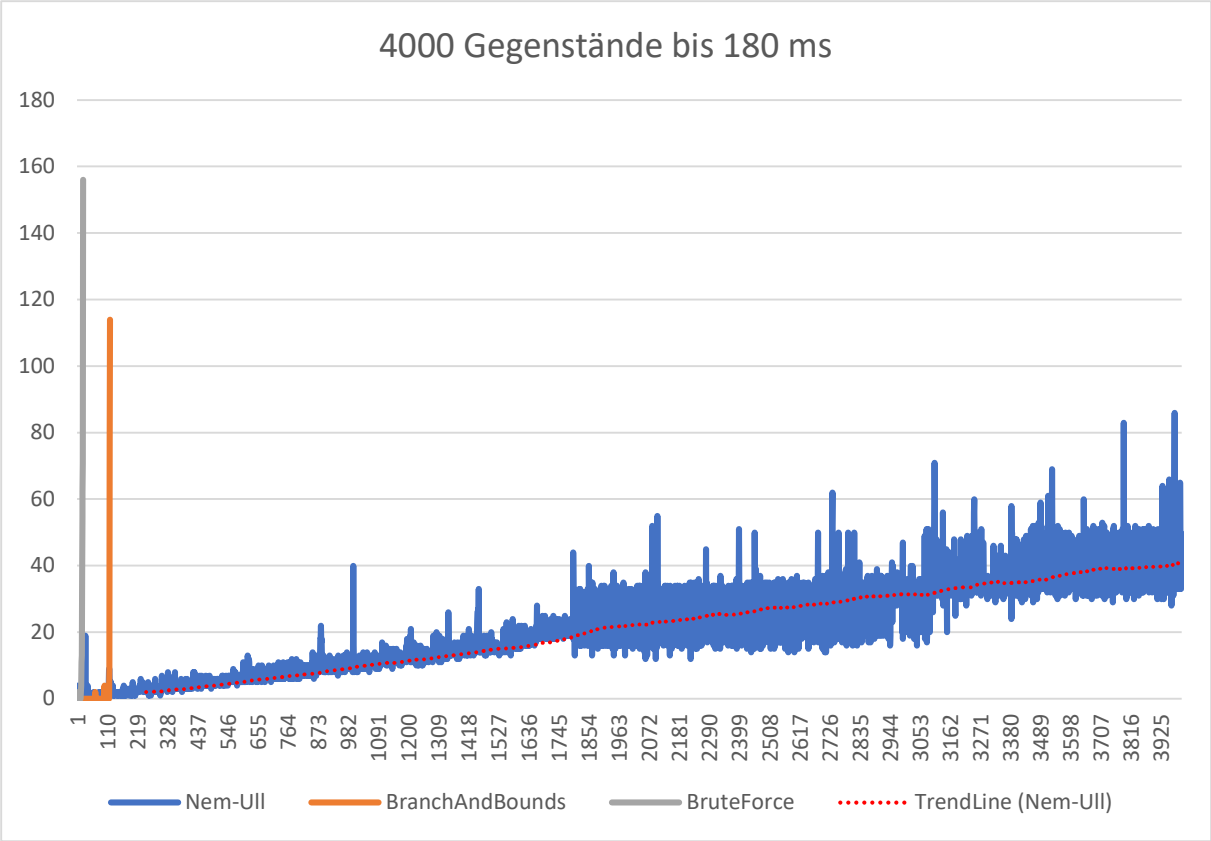
Es gibt einen Raketenstart, bei dem ein Gewicht von 645kg noch frei ist. Es soll eine Kombination aus einer Liste von 8 von Gegenständen gefunden werden, die innerhalb des Gewichtslimits liegt und dabei den höchsten Profit erzieht.

Objekt-Nr.	Gewicht in kg	Profit in 1.000 Euro
1	191	201
2	239	141
3	66	50
4	249	38
5	137	79
6	54	73
7	153	232
8	148	48

Das Gewichtslimit von 645 ist T, das Gewicht der Gegenstände w und der Profit v. Die Anzahl der Gegenstände ist unser n, was hier 8 ist. Es gibt also 256 verschiedene Kombinationen, die es zu prüfen gilt. Aus diesen Kombinationen muss nun die Kombination zurückgeliefert werden, die den höchsten Profit liefert.

Laufzeiten





Die Laufzeiten sind ansteigenden Gegenstand Anzahlen und gleichbleibendem Ziel Gewicht von 6450 KG aufgenommen worden. Die Horizontale Achse zeigt die Gegenstandsanzahl und die Vertikale Achse die Zeit in Millisekunden.

Wie zusehen ist, wachsen die Werte für den Nemhauer-Ullmann-Algorithmus linear an.

Dies entspricht den Erwartungen, da die verwendetet Implementierung eine Laufzeit von $n * T + n$ hat und T statisch ist.

Die Werte für alle Algorithmen wurden auf einem HP DragonFly mit i7-8565U und 16 GB und Windows 11 aufgenommen.

Implementierung und Java Code

Für die Implementierung werden zwei Klassen benötigt.

Die erste Klasse ist CargoItem. Hier werden für das Objekt Name, Gewicht und Profit gespeichert.

Zusätzlich wird eine compareTo Methode benötigt.

```
class CargoItem implements Comparable<CargoItem> {

    int weight, profit;
    String name;

    public CargoItem(int weight, int profit, String name) {
        this.weight = weight;
        this.profit = profit;
        this.name = name;
    }

    public int getWeight() {
        return this.weight;
    }

    public int getProfit() {
        return this.profit;
    }

    public String getName() {
        return this.name;
    }

    public String toString() {
        return this.name + " - weight: " + this.weight + " - profit: " +
this.profit + " - density: " + (this.profit * 1.0 /this.weight);
    }

    @Override
    public int compareTo(CargoItem o) {
        if (o.weight == this.weight) {
            return 0;
        }

        if (o.weight < this.weight) {
            return 1;
        }

        return -1;
    }
}
```

Als zweite Klasse wird die Rucksackproblem benötigt. Es gibt 2 Methoden in Rucksackproblem. Die Methode main erstellt eine Liste von Objekten und setzt ein Gewichtslimit von 645. Danach wird die Knapsack Methode aufgerufen und speichert das Ergebnis. Anschließend werden die Objekte ausgegeben und die Summe des Gewichts und des Profits ausgegeben.

```
public class Rucksackproblem {

    public static void main(String[] args) {

        List<CargoItem> items = new ArrayList<CargoItem>();

        items.add(new CargoItem(191, 201, "Objekt1"));
        items.add(new CargoItem(239, 141, "Objekt2"));
        items.add(new CargoItem(66, 50, "Objekt3"));
        items.add(new CargoItem(249, 38, "Objekt4"));
        items.add(new CargoItem(137, 79, "Objekt5"));
        items.add(new CargoItem(54, 73, "Objekt6"));
        items.add(new CargoItem(153, 232, "Objekt7"));
        items.add(new CargoItem(148, 48, "Objekt8"));

        int weightLimit = 645;

        CargoItem[] knapsack = knapsack(weightLimit, items.toArray(new
CargoItem[items.size()]));

        int summeWeight = 0;
        int summeProfit = 0;

        for (int i = 0; i < knapsack.length; i++) {
            System.out.println(knapsack[i].toString());
            summeProfit += knapsack[i].getProfit();
            summeWeight += knapsack[i].getWeight();
        }

        System.out.println("summeWeight: " + summeWeight);
        System.out.println("summeProfit: " + summeProfit);
    }
```

Die Knapsack-Methode sortiert die Liste von Gegenständen nach Gewicht. Anschließend wird eine Matrix erstellt, die Gewicht*Gegenstandanzahl Felder groß ist.

Anschließend wird pro Gegenstand für jedes KG-Gewicht folgende Logik ausgeführt.

Zuerst wird der Wert für dasselbe Gewicht vom vorherigen Gegenstand übernommen. Anschließend wird getestet, ob das Gewicht des aktuellen Gegenstands bereits größer ist als das Gewicht, in der Variable weightCounter befindet.

Wenn das Gewicht kleiner ist, dann wird getestet, ob der Wert aus der vorherigen Liste, an Position weightCounter – dem Gewicht des aktuellen Gegenstands, plus dem aktuellen Wert größer ist als der zuvor übernommene Wert.

```
    public static CargoItem[] knapsack(int capacity, CargoItem[] items) {

        Arrays.sort(items);

        final int N = items.length;

        // Gewicht/Wert Tabelle erstellen
        int[][] DP = new int[N + 1][capacity + 1];

        // Gegenstände iterieren
        for (int i = 1; i <= N; i++) {
            // Gewicht iterieren
            for (int weightCounter = 1; weightCounter <= capacity;
weightCounter++) {

                // Vorherigen Wert übernehmen
                DP[i][weightCounter] = DP[i - 1][weightCounter];

                    // Testen Gewicht unter Limit
                if (weightCounter >= items[i - 1].getWeight() &&
                    // Testen, ob Ergebnis besser ohne verherigen gegenstand
                    DP[i - 1][weightCounter - items[i - 1].getWeight()] +
items[i - 1].getProfit() > DP[i][weightCounter])
                {
                    DP[i][weightCounter] = DP[i - 1][weightCounter - items[i -
1].getWeight()] + items[i - 1].getProfit();
                }
            }
        }

        int weightCountDown = capacity;
        List<CargoItem> itemsSelected = new ArrayList<CargoItem>();

        for (int i = N; i > 0; i--) {
            if (DP[i][weightCountDown] != DP[i - 1][weightCountDown]) {
                CargoItem itemIndex = items[i - 1];
                itemsSelected.add(itemIndex);
                weightCountDown -= items[i - 1].getWeight();
            }
        }

        return itemsSelected.toArray(new CargoItem[itemsSelected.size()]);
    }
}
```

Zu der hier gezeigten Lösung ist zu sagen, dass diese der Logik des Nemhauser-Ullmann Algorithmus aufbaut, aber keine Implementierung ist, wie Sie in dem „Buch Taschenbuch der Algorithmen“ zu sehen ist.

Die originale Implementierung kann in bestimmten Szenarien (gleiche Profitdichte aller Gegenstände) in eine Laufzeit von 2^n ausarten. Dies ist hier nicht der Fall, da zuerst eine Matrix der Pareto-

Optimalen Punkte erzeugt wird und erst später die Liste der Gegenstände aus dieser Matrix erzeugt wird. Diese Variante garantiert eine Laufzeit von n*T+n. Wichtig zu beachten ist, dass dies die Erstellung der Matrix benötigt und deshalb n+1*T Speicher benötigt. Die kann bei großen n und T Werten problematisch werden.

Zusätzlich gibt es noch zu beachten das der Algorithmus nur Pseudo-Polynomial ist. Dies ist der Fall, da die Laufzeit nicht nur von der Anzahl der Gegenstände, sondern auch von dem Gewichtslimit abhängt. Die Laufzeit desselben Algorithmus mit gleichen Gegenständen wird sich also linear verändern, wenn das Gewichtlimit angepasst wird.

Zusätzlich ist zu beachten, dass nur eine Pareto-Optimale Lösung geliefert. Falls es jedoch mehrere Lösungen gibt, dann wird immer nur die Erste der möglichen Lösungen ausgegeben. Bei der Implementierung, wie sie im Buch Taschenbuch der Algorithmen gezeigt ist, kann es passieren, dass am Ende mehrere Lösungskombinationen vorhanden sind, die alle Pareto-Optimal sind.

Hinweise zur Ausführung:

Die Implementierung des Codes sind in der mitgelieferten JAR-Datei zu finden. Der Code kann direkt ausgeführt werden.

Alle zur Erstellung dieses Seminars genutzten Ressourcen sowie die angepassten Implementierungen zur Zeitmessung sind unter https://github.com/Tabisch/ALDS2324/tree/main/Seminar zu finden.

Zusammenfassung

Die Lösung des 0/1-Knapsackproblems mit Hilfe des Pareto-Optimums ermöglicht es, das Problem in nahezu linearer Zeit bearbeiten zu können. Jedoch ist zu berücksichtigen, dass es keine Möglichkeit gibt, diese Laufzeit zu garantieren.

Es muss daher je nach Implementierung in kaufgenommen werden, dass die Performance massiv degradieren kann oder sich die Performance Garantie durch erhöhten Speicherplatzverbrauch „erkauft“ werden muss.

Quellen

https://www.geeksforgeeks.org/introduction-to-knapsack-problem-its-types-and-how-to-solve-them/

https://www.interviewbit.com/blog/0-1-knapsack-problem/

https://link.springer.com/content/pdf/10.1007/978-3-540-76394-9_41.pdf?pdf=inline%20link

https://www.baeldung.com/cs/knapsack-problem-np-completeness

https://www.youtube.com/watch?v=ujhi2h70qIw

https://www.youtube.com/watch?v=Zz7hmnpOXEI

https://www.youtube.com/watch?v=CiX_eG0dXBo

https://www.mpi-inf.mpg.de/fileadmin/inf/d1/teaching/summer16/random/smoothedanalysis.pdf

https://www.geeksforgeeks.org/branch-and-bound-algorithm/

https://www.youtube.com/watch?v=cJ21moQpofY

https://github.com/williamfiset/Algorithms/blob/master/src/main/java/com/williamfiset/algorithms/dp/Knapsack_01.java

Taschenbuch der Algorithmen S. 391-397

https://www.yumpu.com/de/document/read/5912955/der-nemhauser-ullmann-algorithmus-und-verallgemeine-rungen