Rucksackproblem

Inhaltsverzeichnis

Inhalt

[Einführung 2](#_Toc153842730)

[Erklärung Algorithmus 3](#_Toc153842731)

[Herangehensweise 4](#_Toc153842732)

[Problembeschreibung 5](#_Toc153842733)

[Java Code 6](#_Toc153842734)

[Quellen 10](#_Toc153842735)

# Einführung

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

Es gibt einen Rucksack, der ein Gewichtslimit (T) hat, und einer Menge aus Gegenständen, die ein Gewicht (w) und einen Wert (v) haben. Es wird nun die Kombination an Gegenständen gesucht, die 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.

Ein Bild, das Diagramm, Kreis, Entwurf enthält.

Automatisch generierte BeschreibungAbbildung 1

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

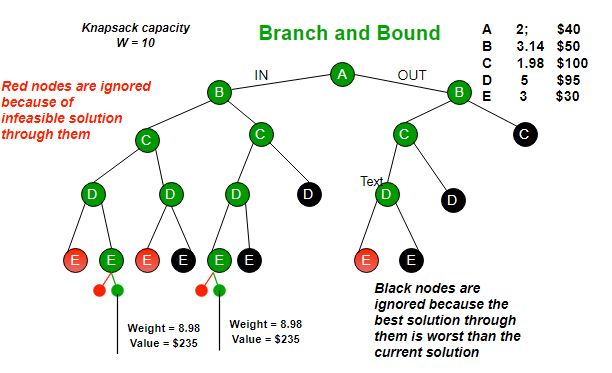
Das 0/1 Knapsack-Problem hat eine Laufzeitkomplexität von wie in der Abbildung zusehen ist.

Das Problem ist NP-Hard, dies bedeutet, dass es weder möglich ist, ein Ergebnis für das Problem in polynomieller Zeit zu finden, noch das es möglich ist, dass Ergebnis in polynomieller Zeit zu verifizieren.

# 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 muss.

Dieses Problem kann mit der Branch and Bound-Methode eingegrenzt werden. Bei diesem Verfahren wird versucht Arme auszuschließen, die keine Möglichkeit haben eine bessere Lösung zu liefern, wie die bisher beste Lösung.



// geeksforgeeks Quelle einfügen

Die gezeigte Grafik erklärt das Branch and Bounds Verfahren.

Die grünen Kreise zeigen Kombinationen, die 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, da der Wert der besten Unterkombination plus den aktuellen Wert des Arms keine Möglichkeit haben über den besten bereits bekannten Wert zu kommen.

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:

Ein Bild, das Reihe, Diagramm, Screenshot, Zahl enthält.

Automatisch generierte Beschreibung

// Grafik überarbeiten

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

Im Beispiel oben ist eine Auflistung aller Kombinationen. 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.

# Problembeschreibung

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

Laufzeiten

# Java Code

Für die Implementierung werden zwei Klassen benötigt.

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

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;

    }

    @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 Mehtode main erstellt eine Liste von Objekten und setzt ein Gewichtslimit von 645. Danach wird die Knapsack Mthode 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-Mehtode Sortiert die Liste von Gegenständen nach Gewicht. Anschließend wird eine Matrix die Gewicht\*Gegenstandanzahl Felder froß ist.

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

Zuerst wird der Wert für das selbe Gewicht vom vorherigen Gegenstand übernommen. Anschließend wird getestet, ob das Gewicht des aktuellen Gegenstands bereits größer ist als das Gewicht, in dem sich der Loop momentan befindet und

    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()]);

    }

}

# 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/