

Brute Force

Seminar 2008

Robert Schilling
(ii5656)

1.	Brute Force	3
1.1.	String Matching	3
1.1.1.	Algorithmus	3
1.1.2.	Pseudocode des Algorithmus	4
1.1.3.	Beispiel für das String Matching	5
1.1.4.	Laufzeitkomplexität	5
1.2.	Closest Pair	6
1.2.1.	Algorithmus	6
1.2.2.	Pseudocode des Algorithmus	7
1.2.3.	Laufzeitkomplexität	7
1.3.	Convex Hull	8
1.3.1.	Definition konvexe Form	8
1.3.2.	Algorithmus	9
1.3.3.	Code für den Algorithmus	11
1.3.4.	Beispiel für Convex Hull	12
1.3.5.	Laufzeitkomplexität	12
1.4.	Traveling Salesman	13
1.4.1.	Algorithmus	13
1.4.2.	Beispiel für Traveling Salesman	13
1.4.3.	Kombinationen des Algorithmus	14
1.5.	Knapsack	15
1.5.1.	Algorithmus	15
1.5.2.	Beispiel für den Algorithmus	16
1.5.3.	Kombinationen	17
1.6.	Job Assignement	18
1.6.1.	Algorithmus	18
1.6.2.	Beispiel des Algorithmus	18
1.6.3.	Kombinationen	18
2.	Literaturverzeichnis	19

1. Brute Force

Das Brute Force Konzept ist ein Straight Forward Konzept, das direkt auf dem Problem basierend implementiert wird. Grundlage für dieses Konzept ist die Tatsache, dass alle Input Element miteinander verglichen werden müssen. Daraus resultieren sehr lange Laufzeiten und meist auch unnötige Berechnungen.

Das Brute Force Konzept ist deswegen auch nur ein erster Entwurf der zum Optimieren benutzt wird.

1.1. String Matching

Der String Matching Algorithmus wird in vielen Texteditoren zum Suchen nach Mustern in einem Text benutzt.

1.1.1. Algorithmus

Es werden zwei Strings angegeben:

- String in dem nach dem Schlüssel gesucht werden soll
(Der String muss grösser sein als der Schlüssel)
- Schlüssel-String
(String der im Hauptstring gesucht wird)

Der Hauptstring wird nun der Länge nach solange durchlaufen, bis der Schlüsselstring gefunden wurde oder das Ende des Hauptstrings (Länge des Hauptstrings – Länge des Schlüssels) erreicht wurde.

1.1.2. Pseudocode des Algorithmus

In der Folgenden Abbildung befindet sich ein Pseudocode für den String Matching Algorithmus:

- „i“ und „j“ sind Laufvariablen
- P ist der Schlüssel
- T ist der Hauptstring
- n ist die Länge des Hauptstrings
- m ist die Länge des Schlüssels

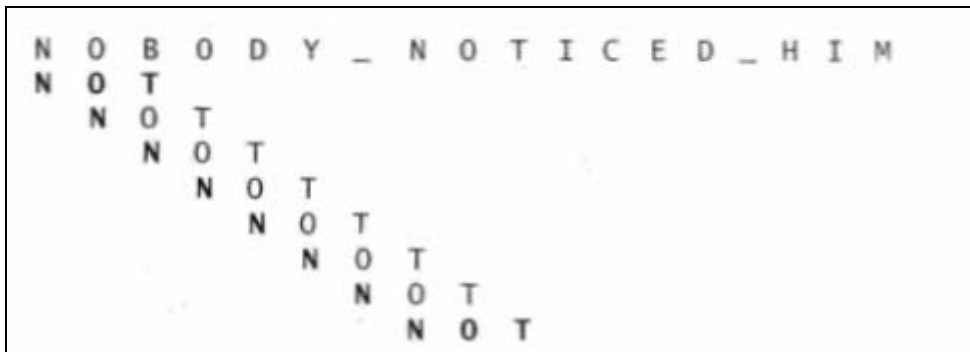
```
for i ← 0 to n - m do
  j ← 0
  while j < m and P[j] = T[i + j] do
    j ← j + 1
  if j = m return i
return -1
```

1.1.2-1 Pseudocode vom String Matching

1.1.3. Beispiel für das String Matching

Für das Beispiel wird der Text „NOBODY NOTICED HIM“ nach dem Muster „NOT“ durchsucht.

In der folgenden Abbildung kann man sehen wie der Text durchlaufen wird und bei dem achten Zeichen des Textes auf eine Übereinstimmung des Musters gestoßen ist.



1.1.3-1 Beispiel für das String Matching

1.1.4. Laufzeitkomplexität

Worst Case: $O(\text{Textlänge} * \text{Musterlänge})$

Beispiel: Text : „ttttttttttt“
Muster : „tte“

Bei diesem Beispiel würde erst beim letzten Vergleich der Indizes vom Muster und Text das nächste Zeichen des Textes verglichen werden. Im Normalfall tritt der Worst Case eher selten auf.

Normal hat man beim String Matching eine Komplexität von $O(\text{Textlänge})!$

1.2. Closest Pair

Mit diesem Algorithmus wird eine Menge an Punkten durchsucht. Es soll das Punktepaar gefunden werden, welches die geringste Distanz zueinander hat in der Menge.

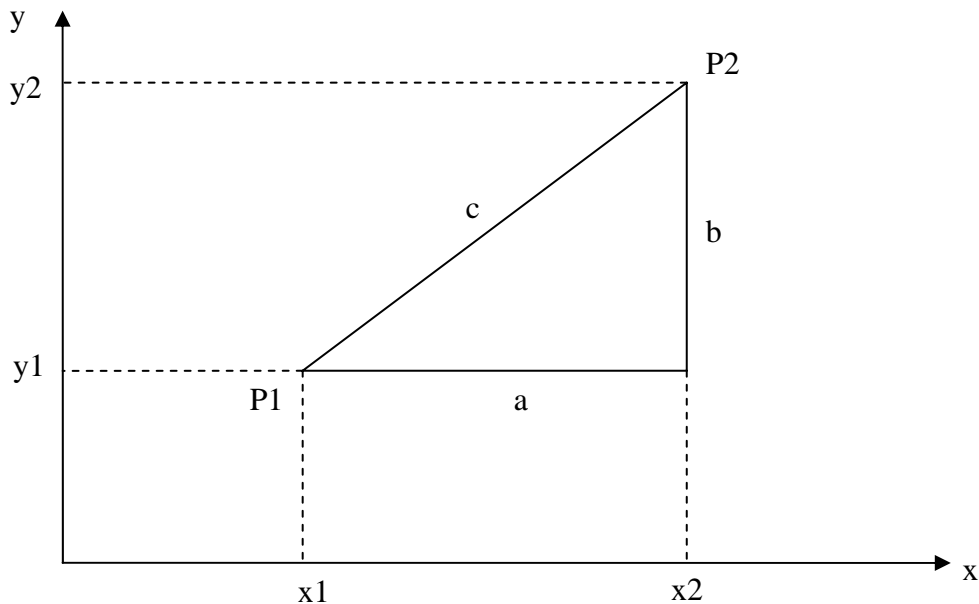
1.2.1. Algorithmus

Eine Menge an Punkten wird als Grundlage für den Algorithmus angegeben.

Die gesamte Menge wird durchlaufen und es muss von jedem Punktepaar die Distanz berechnet werden.

Um die Distanz zwischen 2 Punkten zu berechnen wird der Satz des Pythagoras benutzt:

$$a^2 + b^2 = c^2$$



Aus der Zeichnung und dem Satz des Pythagoras ergibt sich:

- $a = |X_2 - X_1|$
- $b = |Y_2 - Y_1|$
- $c = \sqrt{a^2 + b^2}$

Um den Algorithmus ein wenig schneller zu machen, kann man das ziehen der Wurzel aus $a^2 + b^2$ vernachlässigen und nimmt c^2 als Distanz der Punkte an.

1.2.2.Pseudocode des Algorithmus

Variablen des Pseudocodes :

- „i“ und „j“ sind Laufvariablen für die Punkte.
- min ist die aktuelle kleinste Distanz von zwei Punkten und wird zum initialisieren auf unendlich gesetzt
- dsquare ist die aktuell berechnete Distanz von 2 Punkten
- X_i, X_j, Y_i und Y_j sind die Punktkoordinaten für die Punkte i und j in der Punktmenge

Wenn eine neue minimale Distanz errechnet wurde, werden der Index der beiden Punkte und die Variable dmin neu gesetzt.

```
ALGORITHM BruteForceClosestPoints2 (P)
// Input: A list P of n (n >= 2) points
//       P1 = (x1, y1), ..., Pn = (xn, yn)
// Output: Indices index1 and index2 of the closest
//         pair of points

min = ∞
for i = 1 to n-1 do
  for j = i+1 to n do
    dsquare = (xi - xj)2 + (yi - yj)2
    if dsquare < min
      min = dsquare
      index1 = i
      index2 = j
    end //if
  end //for j
end //for i
return index1, index2
end
```

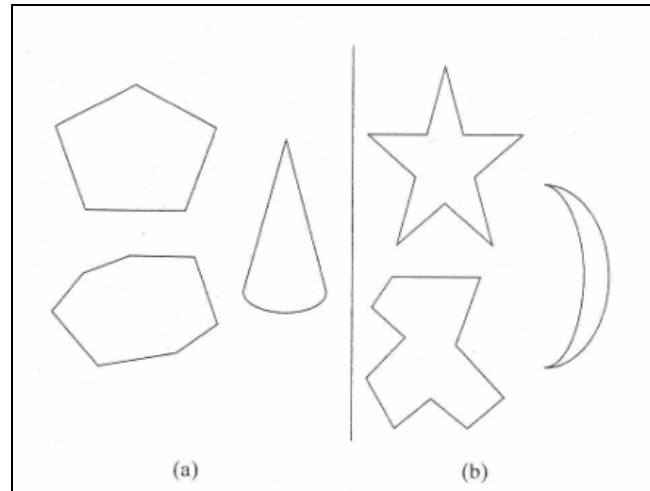
1.2.2-1 Pseudocode vom Closest Pair Algorithmus

1.2.3.Laufzeitkomplexität

Da es zwei Schleifen gibt, die zum Punkte durchlaufen und vergleichen benutzt werden, ist die Komplexität des Closest Pair Algorithmus **O(n²)**.

1.3.Convex Hull

Ähnlich wie der Closest Pair Algorithmus hat man als Ausgangslage eine Menge an Punkten. Mit dieser Menge soll eine konvexe Hülle gebildet werden. In der folgenden Abbildung erkennt man verschiedene Formen, von denen die Formen der Menge a) konvex und in Menge b) nicht konvex sind.



0-1 a) Convexe und b) nicht Convexe Formen

1.3.1.Definition konvexe Form

Bezeichnung für eine Punktmenge, wenn sie zu je zwei Punkten auch deren Verbindungsstrecke enthält.

1.3.2. Algorithmus

Der Algorithmus durchläuft die Punkte der Menge wie beim Closest Pair Algorithmus. Aus jedem Punktepaar wird die allgemeine Geradengleichung aufgestellt:

$$ax + by = c$$

1.3.2-1 allgemeine Geradengleichung

Um die Konstanten a, b und c zu bestimmen benutzt man die Geradennormalform:

$$y = mx + d$$

1.3.2-2 Geradennormalform

Die Steigung m in der Geradennormalform kann man aus dem aktuellen Punktepaar bestimmen:

$$m = \frac{y_i - y_j}{x_i - x_j}$$

1.3.2-3 Steigung von zwei Punkten

Nach Auflösen der allgemeinen Geradengleichung nach y erhält man folgende Gleichung:

$$y = \frac{-a}{b} * x + \frac{c}{b}$$

1.3.2-4 Nach y umgestellte allg. Geradengleichung

Mit einem einfachen Koeffizientenvergleich kann man nun a und b bestimmen:

$$m = \frac{a}{-b}$$

$$a = y_i - y_j$$

$$b = x_j - x_i$$

Um c zu bestimmen setzt man nun a und b in die allgemeine Geradennormalform ein und gibt $x = x_i$ und $y = y_i$ an:

$$c = (y_i - y_j) * x_i + (x_j - x_i) * y_i =$$
$$y_i * x_i - y_j * x_i + y_i * x_j - x_i * y_i = y_i * x_j - x_i * y_j$$

Somit hat man nun die Konstanten a , b und c bestimmt. Um nun mit der allgemeinen Geradennormalform zu ermitteln ob das aktuelle Punktepaar den Rand der Menge bildet muss man alle anderen Punkte in die Gleichung einsetzen. Hierbei gibt es drei Fallunterscheidungen für jeden Punkt:

$$\boxed{a^* x + b^* y - c > 0} \quad \text{Punkt ist auf der rechten Seite der Geraden}$$

$$\boxed{a^* x + b^* y - c = 0} \quad \text{Punkt ist direkt auf der Geraden}$$

$$\boxed{a^* x + b^* y - c < 0} \quad \text{Punkt ist auf der linken Seite der Geraden}$$

Wenn nun alle Punkte auf nur einer Seite oder direkt auf der Geraden sind dann bildet dieses Punktepaar den Rand des konvexen Körpers.

1.3.3.Code für den Algorithmus

```
Point lvP1;
Point lvP2;
Point lvP3;
int a, b, c;

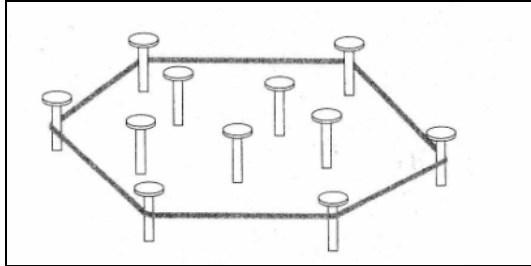
ivFinished = false;
// erste Schleife um vom ersten bis zum vorletzten Punkt durchzulaufen
for (int i = 0; i < ivPoints.size() - 1; i++) {
    // Punkt am Index i ermitteln
    lvP1 = ((DrawPointType) ivPoints.get(i)).getPoint();
    ivIndex&Compared.x = i;
    // zweite Schleife, um die noch verbleibenden Punkte mit dem Punkt
    // an
    // Index "i" zu vergleichen
    for (int j = i + 1; j < ivPoints.size(); j++) {
        ivIndex&Compared.y = j;
        lvP2 = ((DrawPointType) ivPoints.get(j)).getPoint();
        // Berechnung des Convex Hull
        a = lvP2.y - lvP1.y;
        b = lvP1.x - lvP2.x;
        c = (lvP1.x * lvP2.y) - (lvP1.y * lvP2.x);
        int onTheLine = 0;
        int onTheSide = 0;
        int onTheOtherSide = 0;
        int res = 0;
        ivIsConvexHull = false;
        // für das aktuelle Punktepaar müssen die restliche Punkte
        // überprüft werden ob sie nur auf einer Seite liegen
        for (int k = 0; k < ivPoints.size(); k++) {
            if ((k != i) && (k != j)) {
                lvP3 = ((DrawPointType) ivPoints.get(k)).getPoint();
                res = (a * lvP3.x + b * lvP3.y - c);
                if (res == 0) {
                    onTheLine++;
                } else if (res > 0) {
                    onTheSide++;
                } else {
                    onTheOtherSide++;
                }
            }
        }
        // for k schleife end
        ivIsConvexHull = ((ivPoints.size() - 2 - onTheLine) == onTheSide)
            || ((ivPoints.size() - 2 - onTheLine) == onTheOtherSide);

        ivListener.changed();
        try {
            sleep(ivSpeed);
        } catch (InterruptedException e) {
        }
    }
    //for j schleife end
}
//for i schleife end
ivFinished = true;
ivListener.changed();
```

1.3.3-1 Javacode vom Convex Hull Algorithmus

1.3.4. Beispiel für Convex Hull

Mit dem Gummiband Prinzip kann man sehr gut veranschaulichen wie eine konvexe Form entsteht. In der unten zu sehenden Abbildung sind mehrere Nägel abgebildet und wenn man um diese Nägel ein Gummiband spannt bildet sich eine konvexe Hülle in der die äußersten Nägel die Konvexe bilden.



1.3.4-1 Gummiband Prinzip

1.3.5. Laufzeitkomplexität

Laut Pseudocode wird von allen Punktepaaren die allgemein Geradengleichung aufgestellt, was schon mal bedeutet, dass zwei Schleifen durchlaufen werden müssen. Da nun aber auch noch die restlichen Punkte in die Geradengleichung eingesetzt werden, ergibt sich eine Komplexität von $O(n^3)$.

1.4. Traveling Salesman

Bei dem Traveling Salesman Algorithmus sucht man den kürzesten Weg um n verschiedene Orte auf unterschiedlich langen Wegen zu besuchen. Man kann dieses Problem auch mit einer Reiseroutenplanung vergleichen, um ohne viel Aufwand möglichst viele interessante Städte zu besuchen. Wichtig hierbei ist, dass man einen Ausgangspunkt wählt, bei dem man seine Tour startet und beendet.

1.4.1. Algorithmus

Für den Algorithmus hat man n verschiedene Orte und zwischen zwei Orten eine bestimmte Strecke. Nun durchläuft man alle möglichen Kombinationen von Ortswegen ohne zwei Mal einen Ort zu durchlaufen bis man wieder an dem Ausgangspunkt angelangt ist.

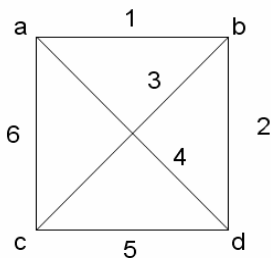
Während dem Durchlaufen werden alle Strecken zusammen addiert und man erhält die Gesamtstrecke die zurückgelegt wurde.

Die kürzeste/n Gesamtstrecke/n ist das Optimum des Algorithmus.

1.4.2. Beispiel für Traveling Salesman

An diesem Beispiel kann man die Vorgehensweise gut veranschaulichen. Es gibt vier Punkte (a, b, c und d) zwischen denen eine bestimmte Strecke liegt.

In dieser Abbildung sind die folgenden Touren optimal:



$$a - b - c - d - a = 1 + 3 + 5 + 4 = 13$$

$$a - b - d - c - a = 1 + 2 + 5 + 6 = 14$$

$$a - c - b - d - a = 6 + 3 + 2 + 4 = 15$$

$$a - c - d - b - a = 6 + 5 + 2 + 1 = 14$$

$$a - d - b - c - a = 4 + 2 + 3 + 6 = 15$$

$$a - d - c - b - a = 4 + 5 + 3 + 1 = 13$$

$$a - b - c - d - a = 1 + 3 + 5 + 4 = 13$$

$$a - d - c - b - a = 4 + 5 + 3 + 1 = 13$$

1.4.2-1 Beispiel für Traveling Salesman

1.4.3. Kombinationen des Algorithmus

Die Kombinationen von diesem Algorithmus sind relativ einfach zu ermitteln.
Bei 5 Städten hat man beim ersten Stadtbesuch 4 Möglichkeiten in die nächste Stadt zu kommen. Bei der nächsten Entscheidung sind es drei und so weiter.

Allgemein kann man daraus schließen, dass es $(n-1)!$ Kombinationen gibt bis man die „beste“ Tour ermittelt hat.

1.5. Knapsack

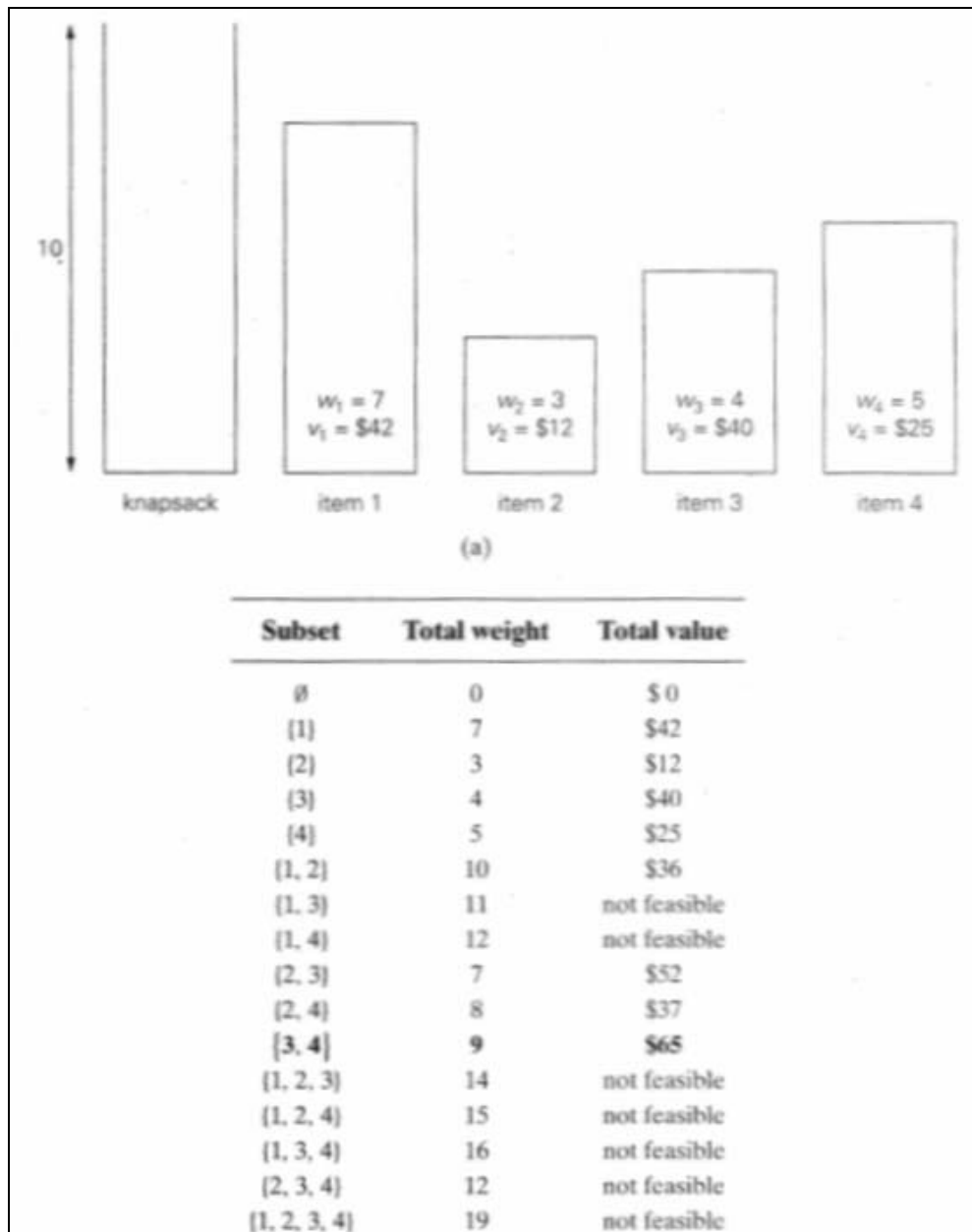
Man nehme an man sei ein Einbrecher in einem Museum und kann nur ein bestimmtes Gewicht an wertvollen Gegenständen in einem Rucksack mitnehmen. In dem Fall würde der Einbrecher versuchen die am meisten Gewinn bringende Kombination an Gegenständen mit sich zu nehmen.

1.5.1. Algorithmus

Nach dem Brute Force Konzept müssen alle möglichen Kombinationen an Gegenständen überprüft werden. Jede Kombination an Gegenständen umfasst ein Gewicht und einen Wert. Der maximale Wert von diesen Kombinationen der auch noch unter dem tragbaren Gewicht liegt ist das Ergebnis dieses Algorithmus.

1.5.2. Beispiel für den Algorithmus

In dem unten abgebildeten Beispiel umfasst der Rucksack (Knapsack) 10 Gewichtseinheiten. Die vier Gegenstände (Items) werden mit jeder Kombination auf Gewicht und Wert verglichen. In diesem Beispiel bringen die Gegenstände 3 und 4 am meisten Gewinn.



1.5.2-1 Knapsack Beispiel

1.5.3. Kombinationen

Die Kombinationen der Gegenstände die man in seinem Rucksack mitnehmen kann sind zweier Potenzen von der Anzahl an Gegenständen selbst.

Allgemein gilt: 2^n .

1.6. Job Assignment

Ein Betrieb steht vor der Entscheidung n Jobs zu besetzen und hat die Möglichkeit n Personen dafür einzuteilen. Die unterschiedlichen Arbeitskräfte bekommen allerdings für denselben Job unterschiedlichen Lohn. Aus diesem Grund gibt es den Job Assignment Algorithmus, mit dessen Hilfe man die minimalste Lohnverteilung für die Arbeitskräfte ermitteln kann.

1.6.1. Algorithmus

Genau wie bei dem Traveling Salesman und Knapsack Algorithmus werden alle Kombinationen der Jobverteilungen aufgestellt.

Die für den Betrieb wirtschaftlichste Lösung ist Ergebnis des Algorithmus.

1.6.2. Beispiel des Algorithmus

	Job 1	Job 2	Job 3
Person 1	5	6	7
Person 2	2	3	4
Person 3	4	2	1

$$\langle 1, 2, 3 \rangle = 5 + 3 + 1 = 9$$

$$\langle 1, 3, 2 \rangle = 5 + 4 + 2 = 11$$

$$\langle 2, 1, 3 \rangle = 6 + 2 + 1 = 9$$

$$\langle 2, 3, 1 \rangle = 6 + 4 + 4 = 14$$

$$\langle 3, 1, 2 \rangle = 7 + 2 + 2 = 11$$

$$\langle 3, 2, 1 \rangle = 7 + 3 + 4 = 14$$

$$\langle 2, 1, 3 \rangle = 6 + 2 + 1 = 9$$

$$\langle 1, 2, 3 \rangle = 5 + 3 + 1 = 9$$

1.6.2-1 Beispiel Tabelle für das Job Assignment

1.6.3. Kombinationen

Die Anzahl der Kombinationen ist ähnlich wie die beim Traveling Salesman.

Einziger Unterschied bei den beiden Algorithmen ist, dass der Anfangspunkt für die Kombinationen bei dem Job Assignment frei gewählt werden kann.

Deswegen sind die Anzahl der Kombination : $n!$

2. Literaturverzeichnis

Anany Levitin: *Introduction to the Design and Analysis of Algorithms*, Addison-Wesley 2006, ISBN 0-321-36413-9

Anmerkung:

Weitere vertiefende Informationen habe ich dem Internet entnommen. Sehr hilfreich war bei der Ausarbeitung „de.wikipedia.org“ sowie einige Mathematikforen.