Algorithmen: Optimierungsalgorithmen

- Prinzip Kombinatorischer Optimierungsalgorithmen verstehen und anwenden können
- Den Baum aller Möglichkeiten aufbauen können
- Die Optimierungsstrategien *Backtracking* und *Branch and Bound* anhand des *Rucksackproblems* anwenden können
- Unterschiede, Vor- und Nachteile obiger Optimierungsstrategien erkennen
- Verstehen wie das *Traveling Salesman-Problem* anhand von Permutationen gelöst werden kann
- Verstehen wie die Enumeration beim Traveling Salesman-Problem begrenzt werden kann
- Den *A*-Algorithmus zu Wegesuche* in seinen Varianten verstehen
- Vorgestelle Optimierungsalgorithmen implementieren können
- Die Java-Datenstrukturen `EnumSet` und `HashSet` verwenden können

Kombinatorische Optimierungsalgorithmen arbeiten nach dem Prinzip *„alle Möglichkeiten erproben“*

Einsatzgebiete	Operations Research, Netzplantechnik, optimierende Compiler
Typische Problemstellungen	Rucksackproblem, Traveling Salesman-Problem, Wegesuche

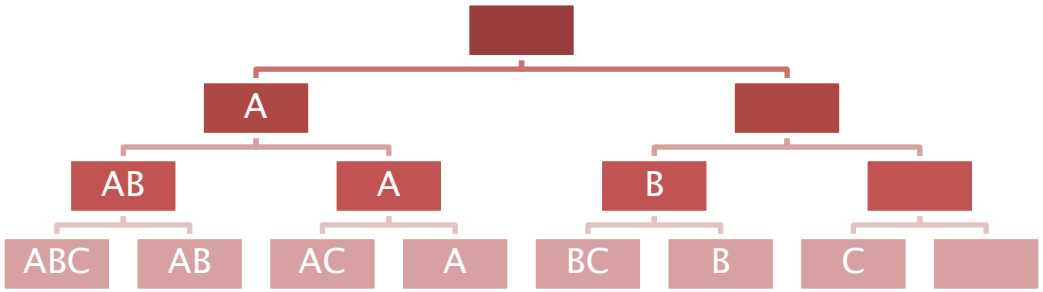
Einführende Problemstellung

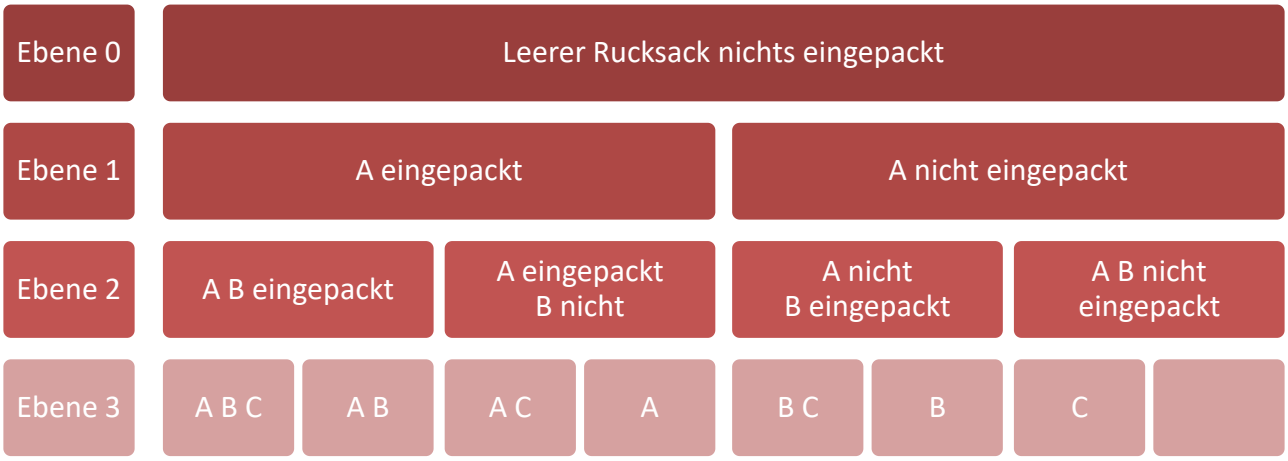
> *Gegeben seien drei Dinge A, B und C die in einen Rucksack gepackt werden sollen. Welche verschiedenen Möglichkeiten gibt es, den Rucksack mit den Dingen zu füllen?*

Annahme: Der Rucksack habe beliebig viel Platz

Baum aller Möglichkeiten

⬁**Links** Ding einpacken **Rechts** Ding nicht einpacken⬂





Frage: Wie viele Endknoten gibt es im Baum aller Möglichkeiten mit n Ebenen, wie viele insgesamt?

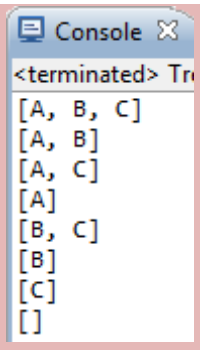
```
import java.util.EnumSet;

public class TreeOfAllPossibilities
{
  public enum Item {A, B, C}

  public static void main(String[] args) {
    EnumSet<Item> items = EnumSet.allOf(Item.class);
    EnumSet<Item> content = EnumSet.noneOf(Item.class);
    addToBag(content, items);
  }

  public static void addToBag(
    EnumSet<Item> content, EnumSet<Item> items) {
    if (items.isEmpty())
     System.out.println(content);
    else {
     Item item = items.iterator().next();
     EnumSet<Item> itemsWithRemovedItem = EnumSet.copyOf(items);
     itemsWithRemovedItem.remove(item);
     EnumSet<Item> contentWithAddedItem = EnumSet.copyOf(content);
     contentWithAddedItem.add(item);

     addToBag(contentWithAddedItem, itemsWithRemovedItem);
     addToBag(content, itemsWithRemovedItem);
    }
  }
}
```



Das Rucksackproblem

Gegeben seien n verschiedene Dinge mit Gewichten (weight) w_j und Werten (value) v_j und ein Rucksack, der nur bis zu einem Grenzwert t gefüllt werden darf.
In den Rucksack sollen solche Dinge eingepackt werden, damit er einen möglichst hohen Wert enthält.

Beispiel

n = 5, t = 38

j	0	1	2	3	4
w_j	8	16	21	17	12
v_j	8	14	16	11	7

Dinge sind nach ihrem relativen Wert geordnet:

$$\frac{v_j}{w_j} \geq \frac{v_{j+1}}{w_{j+1}}$$

Beste Lösung {1, 2} mit Gewicht 37 und Wert 30

Deklaration des Knotens

```
public class TreeNode
{
  private HashSet<Integer> content = new HashSet<Integer>();
  private int weight = 0;
  private int value = 0;
}
```

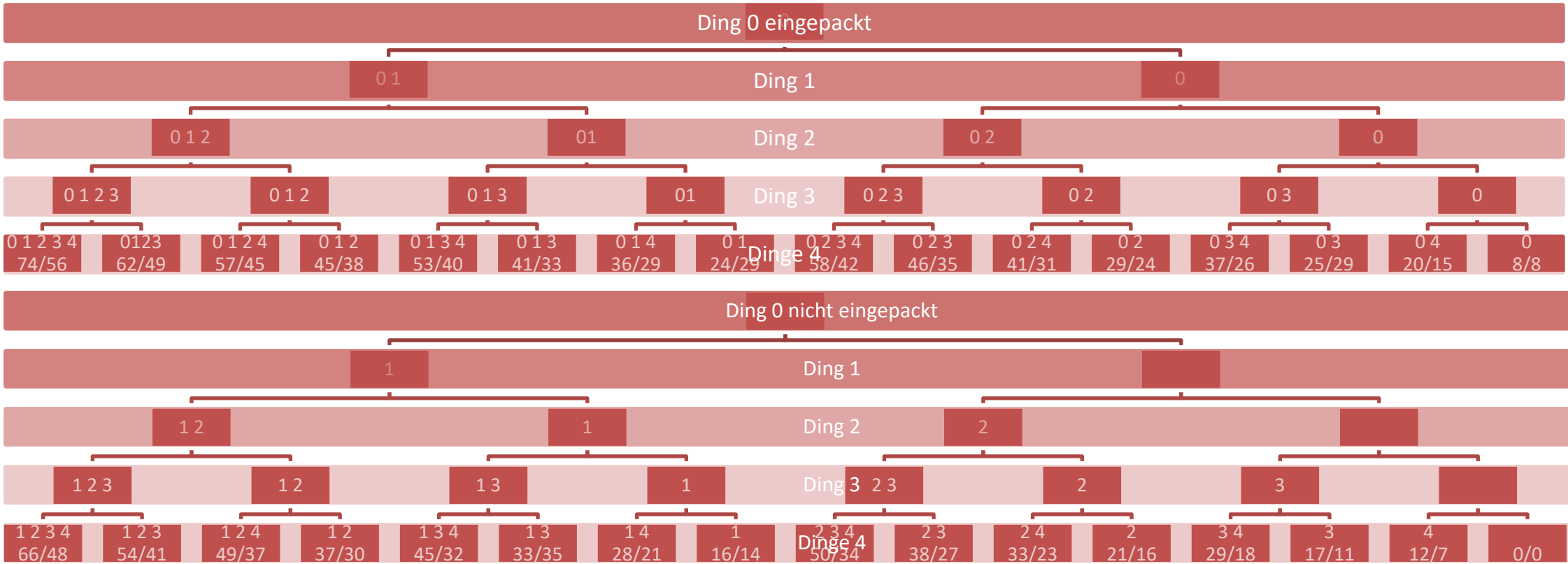
Baum aller Möglichkeiten

```
private static final int T = 38;

private static final int[][] ITEMS =
  {{8, 8}, {16, 14}, {21, 16}, {17, 11}, {12, 7}};

public static void main(String[] args) {
  allPossibilities(new TreeNode(), 0);
}

public static void allPossibilities(TreeNode node, int level) {
  if (level == items.length)
    System.out.println(node);
  else {
    TreeNode nodeLeft = null;
    TreeNode nodeRight = null;
    nodeLeft = (TreeNode)node.clone();
    nodeLeft.content.add(level);
    nodeLeft.weight+=ITEMS[level][0];
    nodeLeft.value+=ITEMS[level][1];
    allPossibilities(nodeLeft, level + 1);
    nodeRight = (TreeNode)node.clone();
    allPossibilities(nodeRight, level + 1);
  }
}
```




```
[0, 1, 2, 3, 4] w = 74 v = 56
[0, 1, 2, 3] w = 62 v = 49
[0, 1, 2, 4] w = 57 v = 45
[0, 1, 2] w = 45 v = 38
[0, 1, 3, 4] w = 53 v = 40
[0, 1, 3] w = 41 v = 33
[0, 1, 4] w = 36 v = 29
[0, 1] w = 24 v = 22
[0, 2, 3, 4] w = 58 v = 42
[0, 2, 3] w = 46 v = 35
[0, 2, 4] w = 41 v = 31
[0, 2] w = 29 v = 24
[0, 3, 4] w = 37 v = 26
[0, 3] w = 25 v = 19
[0, 4] w = 20 v = 15
[0] w = 8 v = 8
[1, 2, 3, 4] w = 66 v = 48
[1, 2, 3] w = 54 v = 41
[1, 2, 4] w = 49 v = 37
[1, 2] w = 37 v = 30
[1, 3, 4] w = 45 v = 32
[1, 3] w = 33 v = 25
[1, 4] w = 28 v = 21
[1] w = 16 v = 14
[2, 3, 4] w = 50 v = 34
[2, 3] w = 38 v = 27
[2, 4] w = 33 v = 23
[2] w = 21 v = 16
[3, 4] w = 29 v = 18
[3] w = 17 v = 11
[4] w = 12 v = 7
[] w = 0 v = 0
```

Console

<terminated> Backtracking [Java Appli

```
[0, 1, 4] w = 36 v = 29
[0, 1] w = 24 v = 22
[0, 2] w = 29 v = 24
[0, 2, 4] w = 33 v = 23
[0, 2] w = 21 v = 16
[0, 3, 4] w = 37 v = 26
[0, 3] w = 25 v = 19
[0, 4] w = 20 v = 15
[0] w = 8 v = 8
[1, 2] w = 37 v = 30
[1, 2, 4] w = 41 v = 31
[1, 2] w = 29 v = 24
[1, 3, 4] w = 45 v = 32
[1, 3] w = 33 v = 25
[1, 4] w = 28 v = 21
[1] w = 16 v = 14
[2, 4] w = 33 v = 23
[2] w = 21 v = 16
[3, 4] w = 29 v = 18
[3] w = 17 v = 11
[4] w = 12 v = 7
[] w = 0 v = 0
```

```
[0, 1, 4] w = 36 v = 29
[0, 2] w = 29 v = 24
[1, 2] w = 37 v = 30
[1, 3] w = 33 v = 25
```

Schranke | node.weight ≤ t | Alle Möglichkeiten

Backtracking

Zulässige Knoten: `node.weight ≤ t`

Unzulässige Knoten: `node.weight > t`

Beste Lösung: Jener zulässige Endknoten mit dem größten Wert

Im Baum arbeitet man sich so weit nach links unten, wie es möglich ist (bei einem unzulässigen Knoten geht man nicht weiter). Sodann geht man zurück, bis Abzweigung nach rechts unten gefunden wird, usw.

Problem:
Teilbäume werden sehr lange verfolgt, auch dann, wenn durch Auslassen von einem oder mehreren Dingen mit hohem Wert überhaupt keine Chance mehr besteht, das bisher beste Ergebnis zu übertreffen

Lösung:

- Für jeden Knoten wird *Schranke* ermittelt, die besagt, welchen besten Wert dieser Teilbaum noch erreichen kann
- Wert von *Schranke* ermitteln, indem ab Knoten alles eingepackt wird, was man ab diesem noch einpacken darf
- Kontrolle des Teilbaumes abbrechen, wenn bisher bestes Ergebnis besser als *Schranke*
- *Schranke* wird um den Wert des Knotens reduziert, wenn man auf ihn verzichtet

Branch and Bound

Hier wird nicht sofort zu den Endknoten durchgebrochen, sondern man bleibt bei mittleren Knoten stehen: *Menge der aktiven Knoten*

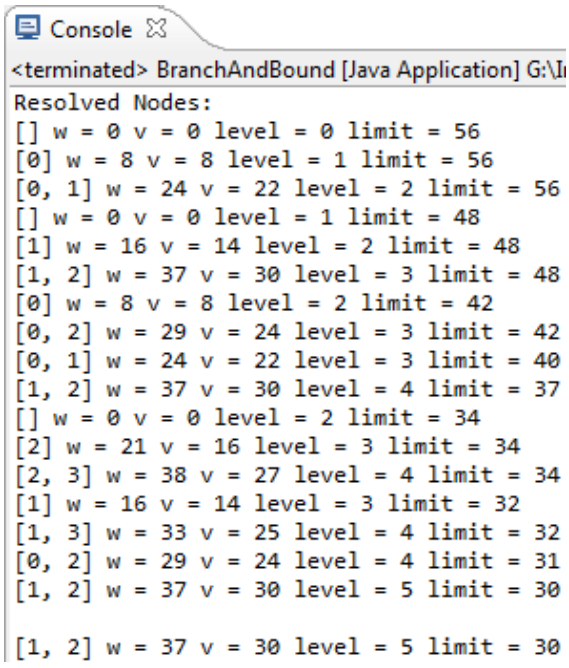
In jedem Knoten wird *Schranke* abgelegt. Je größer Schranke des Knoten, desto größer ist die Hoffnung, von ihm aus einen sehr guten Endknoten zu erreichen

Daher: vom aktiven Knoten mit größter Schranke weitergehen

Sobald man den ersten Knoten der letzten Ebene aus der Menge herausnimmt, gelangt man wie bei *Backtracking* zu einer potentiellen Lösung. Zugleich weist dieser gemäß der Sortierung der aktiven Knoten die höchste Schranke auf, was am wenigsten an Wertverzicht bedeutet. In anderen Worten ausgedrückt hat er den höchstmöglichen Wert und stellt dabei automatisch die optimale Lösung dar.

Vorteil:
Unsinnige Teilbäume werden in der Verfolgung weit zurückgedrängt, sodass wahrscheinlich weniger Knoten untersucht werden als beim Backtracking

Nachteil:
Liste der aktiven Knoten muss vollständig gespeichert werden … Speicherplatzaufwand!!! (Geeignete Datenstruktur Heap)



Traveling Salesman-Problem

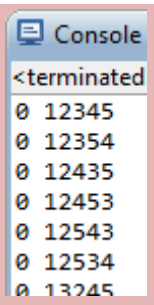
Gegeben ist ein gerichteter Graph mit n + 1 Knoten (Städten) und bewerteten Kanten (Straßen). Es ist die kürzeste (kostengünstigste) Rundreise gesucht, die genau einmal durch jede Stadt führt.

Lösung

- Alle Permutationen von 1 bis n bilden, jede entspricht einer Rundreise (z.B. 0 54231)
- Durch Permutationen werden keine Städte doppelt angefahren

1. Schritt: Rundreisen aufbauen

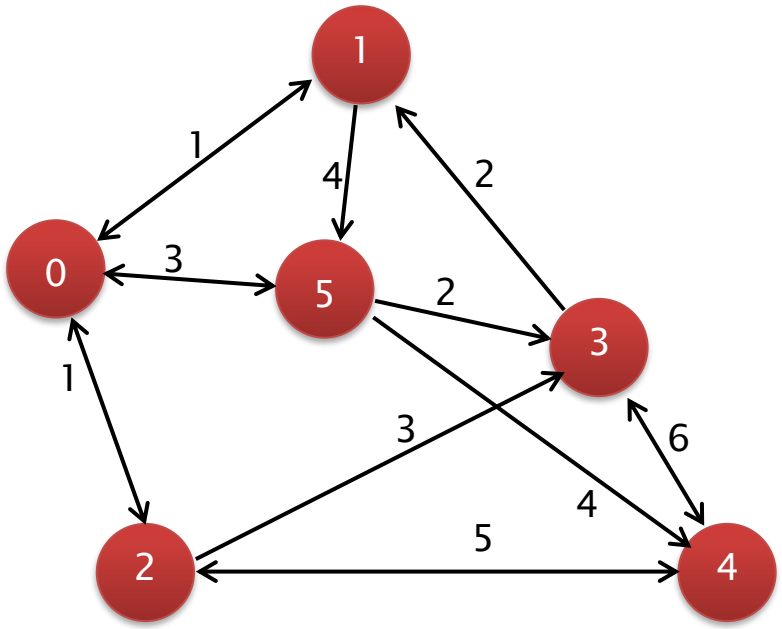
```
public static int[] trip = {0, 1, 2, 3, 4, 5};
public static void main(String[] args) {
  permute(1);
}
public static void permute(int pos) {
  if (pos == trip.length - 1)
    outPerm(trip);
  else {
    permute(pos + 1);
    for (int i = pos + 1; i < trip.length; i++) {
     int h = trip[pos]; trip[pos] = trip[i]; trip[i] = h;
     permute(pos + 1);
     h = trip[i]; trip[i] = trip[pos]; trip[pos] = h;
    }
  }
}
```



- In globaler Variable `trip` wird Permutation aufgebaut

2. Schritt: Begrenzung der Enumeration

- Unzulässige Rundreisen (Permutationen) und Rundreisen die schlechter als bisher beste Rundreise sind nicht weiter verfolgen



	0	1	2	3	4	5
0	-	1	1	MAX_VALUE	MAX_VALUE	3
1	1	-	MAX_VALUE	MAX_VALUE	MAX_VALUE	4
2	1	MAX_VALUE	-	3	5	MAX_VALUE
3	MAX_VALUE	2	MAX_VALUE	-	6	MAX_VALUE
4	MAX_VALUE	MAX_VALUE	5	6	-	MAX_VALUE
5	3	MAX_VALUE	MAX_VALUE	2	4	-

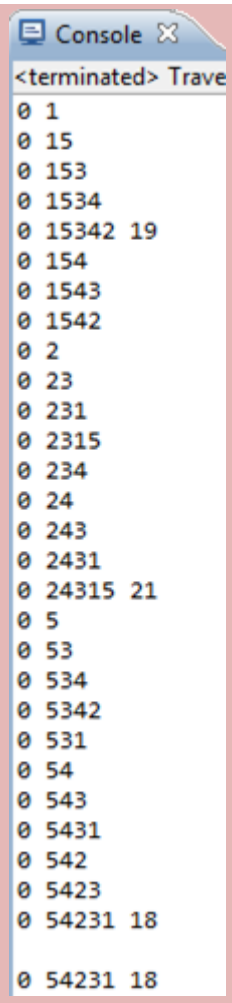
- Kostenmatrix `costMatrix[i][k]` = MAX_VALUE falls kein Weg von `i` nach `k` führt
- Globale Variable `bestTrip` speichert bisher beste Rundreise
- `bestCosts` speichert die Kosten von bisher besten Rundreise

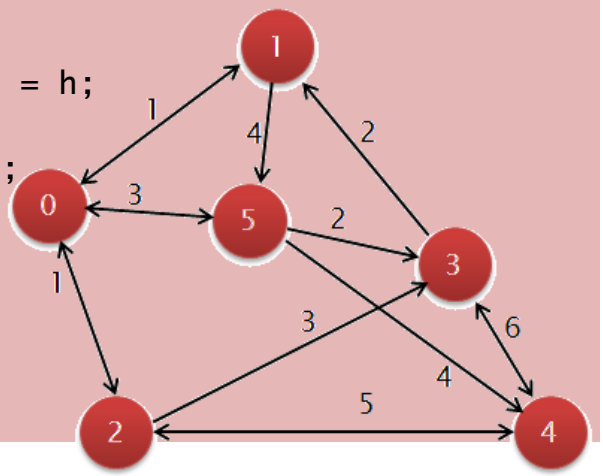
```
public static int[] trip = {0, 1, 2, 3, 4, 5};

public static int[] bestTrip = null;
public static int bestCosts = MAX_VALUE;

public static void main(String[] args) {
  traveling(1,0);
}

public static void traveling(int pos, int costs) {
  int newCosts = 0;
  if (pos == N - 1) {
    // Rundreise über trip[5] zurück nach trip[0]
    newCosts = costs +
      COST_MATRIX[trip[pos - 1]][trip[pos]] +
      COST_MATRIX[trip[pos]][trip[0]];
    if (newCosts < bestCosts) {
      // Bessere Rundreise gefunden
      bestTrip = trip.clone();
      bestCosts = newCosts;
    }
  } else {
    int h = trip[pos];
    for (int i = pos; i < N; i = i + 1) {
      newCosts = costs +
        COST_MATRIX[trip[pos - 1]][trip[i]];
      // Weitermachen?
      if (newCosts < bestCosts) {
        trip[pos] = trip[i]; trip[i] = h;
        outPerm(trip,pos + 1);
        traveling(pos + 1, newCosts);
        trip[i] = trip[pos];
      }
    }
    trip[pos] = h;
  }
}
```





Wegesuche durch den A*-Algorithmus

In einem gerichteten Graphen sind alle Kanten bewertet (>0). Gesucht ist der kürzeste Weg von einem gegebenen Startort S zu einem gegebenen Zielort Z.

Open-Liste enthält alle *aktiven Knoten*:

- Ein Knoten in Open-Liste steht für einen Weganfang von Startort S zu einem bestimmten Ort K im Graphen
- Zu jedem Knoten der Open-Liste ist ein Weg vom Startort S nach K bekannt. Das ist vielleicht der Anfang des gesuchten kürzesten Weges von S nach Z

Knoten auflösen bedeutet:
Knoten der Open-Liste wird in Open-Liste durch seine Nachfolger ersetzt, d. h. ein Weganfang S nach K wird durch längere Weganfänge S nach Nachfolger(K) ersetzt. Kürzesten Weg gefunden, wenn Z aufgelöst wird

Achtung: Nur sinnvolle Weganfänge weiterverfolgen

Sinnvoll: Löse jenen Knoten auf, für den die Summe aus Weganfang S nach K plus *untere Schranke* für den Wegrest K nach Z am kleinsten ist (Luftlinie)

```
public class Node
{
  // Entfernung S nach K
  int distance = 0;
  // Untere Schranke für die Entfernung K nach Z
  int limit = 0;
  // Vorgängerknoten für Rückwegbestimmung
  Node previous = null;
}
```

Bemerkungen:

- Von jenen Knoten mit gleichem `distance + limit` jenen auflösen, der größere `distance` hat (man ist in der Regel nicht mehr weit entfernt von Z)
- `limit = 0` für alle Knoten bedeutet: Es wird jener Knoten zuerst aufgelöst, der näher bei S liegt
- Wenn Ort K mehrmals in Knoten der Open-Liste vorkommt, dann schadet das „theoretisch“ nichts, aber wenn die Unterschiede bei `distance` klein sind, dann ziehen die anderen bald nach; bei den Nachfolgern wiederholt sich alles ... Aufwand!!!
- Wenn im Graphen Kanten nicht bewertet sind (Entfernung zw. Knoten kann 0 sein), dann besteht Gefahr evtl.er Endlosschleifen
- Es kann abgebrochen werden, wenn Z aufgelöst wird