Programmierkurs in C

Universität Augsburg

Wintersemester 2017/2018

Prof. Dr. Robert Lorenz

Lehrprofessur für Informatik

Tag 5: Optimierungsprobleme, Backtracking

Lernziele

Das **Backtracking-Verfahren** zur Lösung von **Optimierungsproblemen** verstehen und anwenden können

Das Verfahren durch **Branch and Bound Technik** verbessern können

Optimierungsprobleme

<u>Optimierungsprobleme</u>

sind Probleme, bei denen

aus einer Menge von möglichen **(zulässigen) Lösungen**

eine **optimale Lösung** bestimmt werden soll

<u>Optimalität</u>

heißt i.d.R., dass eine **Zielfunktion minimiert/maximiert** wird

(u.U. unter sog. **Nebenbedingungen**)□

Beispiele

Rucksackproblem:

<u>Gegeben:</u>
Eine obere Schranke $\mathbf{y}$
Menge $\mathbf{G}$ von $\mathbf{n}$ Gegenständen mit Gewichten $\mathbf{x}_1, \ldots, \mathbf{x}_n$

<u>Gesucht:</u>
Teilmenge (wird in den Rucksack gepackt),
so dass das Gesamtgewicht aller enthaltenen Elemente maximal und kleinergleich $\mathbf{y}$ ist.

Beispiele

Rucksackproblem:

Gegeben:
Eine obere Schranke $\mathbf{y}$
Menge $\mathbf{G}$ von $\mathbf{n}$ Gegenständen mit Gewichten $\mathbf{x_1,...,x_n}$

Gesucht:
Teilmenge (wird in den Rucksack gepackt),
so dass das Gesamtgewicht aller enthaltenen Elemente maximal und kleinergleich $\mathbf{y}$ ist.

Zulässige Lösung: $X \subseteq G$
Zielfunktion: $\text{summe}\{ x \mid x \in X \}$
Nebenbedingung: $\text{summe}\{ x \mid x \in X \} <= y$

Beispiele

Rucksackproblem, konkretes Beispiel:

Gegeben:
Obere Schranke: $y = 30$
3 Gegenstände mit Gewichten: $x_1=12$, $x_2=15$, $x_3=16$

Zulässige Lösungen (alle Teilmengen):
{ }, {12}, {15}, {16}, {12,15}, {12,16}, {15,16}, {12,15,16}

Optimale Lösung:
{12,16}, summe({12,16})=28

Beispiele

8-Damen-Problem:

Gegeben:
8 Damen und ein Schachbrett

Gesucht:
Verteilung der Damen auf dem Schachbrett, so dass sich keine zwei dieser Damen gegenseitig bedrohen

Beispiele

8-Damen-Problem:

Gegeben:
8 Damen $d_1, \ldots, d_8$ und ein Schachbrett

Gesucht:
Verteilung `V` der Damen auf dem Schachbrett, so dass sich keine zwei dieser Damen gegenseitig bedrohen

Zulässige Lösung:
$V = ((zeile_1, spalte_1), \ldots, (zeile_8, spalte_8))$

Zielfunktion:
`f(V) = 0, falls sich Damen bedrohen, sonst = 1`

Beispiele

8-Damen-Problem, konkretes Beispiel:

<u>Zulässige Lösung</u>:

1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0

`f(V)=0`

Beispiele

8-Damen-Problem, konkretes Beispiel:

Zulässige Lösung:

1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0

`f(V)=0`

Beispiele

8-Damen-Problem, konkretes Beispiel:

Zulässige Lösung:

1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0

`f(V)=0`

Beispiele

8-Damen-Problem, konkretes Beispiel:

Zulässige Lösung:

1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0

f(V)=0

Beispiele

8-Damen-Problem, konkretes Beispiel:

Zulässige Lösung:

1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0

$f(V)=0$

Beispiele

8-Damen-Problem, konkretes Beispiel:

Zulässige Lösung:

```
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0
```

`f(V)=0`

Beispiele

8-Damen-Problem, konkretes Beispiel:

Zulässige Lösung:

```
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
```

```
f(V)=0
```

Beispiele

8-Damen-Problem, konkretes Beispiel:

Zulässige Lösung:

1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1

`f(V)=0`

Beispiele

8-Damen-Problem, konkretes Beispiel:

Zulässige Lösung:

1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0

`f(V)=0`

Beispiele

8-Damen-Problem, konkretes Beispiel:

<u>Zulässige Lösung</u>:

```
1  0  0  0  0  0  0  0
1  0  0  0  0  0  0  0
1  0  0  0  0  0  0  0
1  0  0  0  0  0  0  0
1  0  0  0  0  0  0  0
1  0  0  0  0  0  0  0
0  1  0  0  0  0  0  0
0  1  0  0  0  0  0  0
```

`f(V)=0`

Beispiele

8-Damen-Problem, konkretes Beispiel:

Zulässige Lösung:

1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	0

`f(V)=0`

Beispiele

8-Damen-Problem, konkretes Beispiel:

Zulässige Lösung:

1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0
0	0	0	1	0	0	0	0

$f(V)=0$

Beispiele

8-Damen-Problem, konkretes Beispiel:

(eine) Optimale Lösung:

0	0	0	0	0	1	0	0
0	0	0	1	0	0	0	0
0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	1
0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0

f(V)=1

Beispiele

Travelling-Salesman-Problem (TSP):

<u>Gegeben:</u>
`n` Städte und ihre Entfernungen voneinander

<u>Gesucht:</u>
Kürzeste Rundreise durch diese Städte

Beispiele

Travelling-Salesman-Problem (TSP):

Gegeben:
`n` Städte und ihre Entfernungen `d[1][1],…,d[1][n],…, d[n][1],…,d[n][n]` voneinander `(d[i][i]=0, d[i][j]=d[j][i])`

Gesucht: Kürzeste Rundreise `R` durch diese Städte

Zulässige Lösungen:
`R = (i[1],…,i[n])` (Rundreise `i[1]`→…→`i[n]`→`i[1]`)

Zielfunktion:
`f(R) = d[i[1]][i[2]] + … + d[i[n-1]][i[n]] + d[i[n]][i[1]]`

Beispiele

Travelling-Salesman-Problem (TSP), konkretes Beispiel:

Gegeben:
4 Städte und ihre Entfernungen voneinander:
`d[1][2]=10,d[2][3]=10,`
`d[3][4]=10,d[1][4]=10,`
`d[1][3]=15,d[2][4]=15`

Zulässige Lösungen:
`(1,2,3,4), (1,2,4,3), (1,3,2,4), (1,4,2,3),…`

Optimale Lösung:
`f(1,2,3,4) = 40`

Systematisches Probieren

Systematisches Probieren basiert auf der Annahme:

Kenntnis / Erzeugbarkeit aller zulässigen Lösungen

Vorgehensweise:

Systematisch alle möglichen zulässigen Lösungen aufzählen

Backtracking

<u>Backtracking: Spezielle Form des systematischen Probierens</u>

<u>Grundkonzept:</u>

Gesamter Lösungsraum ist baumartig angeordnet.

<u>Backtracking-Algorithmus durchläuft diesen Baum in bestimmter Reihenfolge:</u>

Ausgehend von einem Knoten des Baumes wird in rekursiver Weise ein Nachfolgeknoten ausprobiert.

Dies wird wieder rückgängig gemacht und dann ein anderer Nachfolgeknoten ausprobiert.

Backtracking - Rucksackproblem

Eingabe:

$\mathbf{y > 0}$

$\mathbf{x_1, ..., x_n}$ mit $\mathbf{g(x_i) > 0}$ für jedes $\mathbf{i}$

Ausgabe:

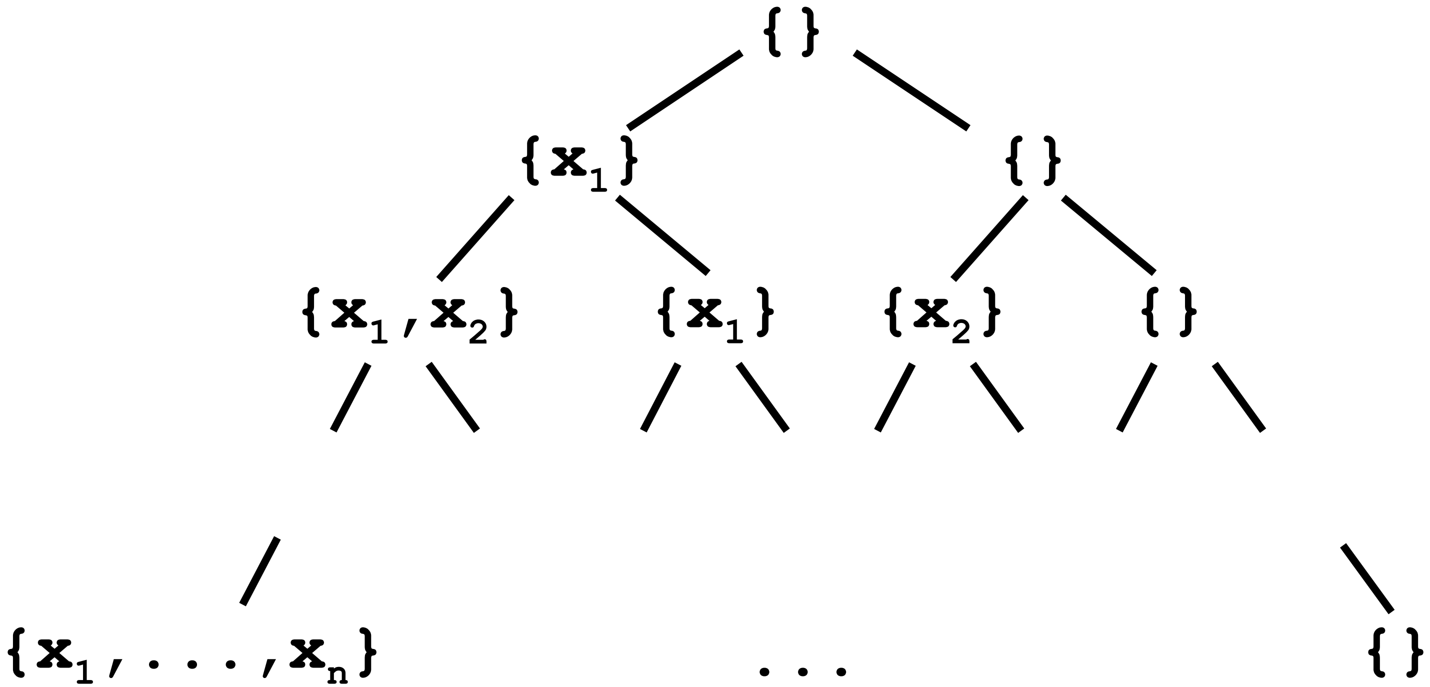
$\mathbf{M} \subseteq \mathbf{\{x_1, ..., x_n\}}$ mit

$\mathbf{gewicht(M) = \Sigma\ g(x) <= y}$ und

$\mathbf{gewicht(M)}$ maximal

Backtracking - Rucksackproblem

Lösungsbaum zur Darstellung aller 2^n Teilmengen von $\{x_1, ..., x_n\}$ (jedes Blatt entspricht einer Teilmenge):



Backtracking - Rucksackproblem

Vorbereitungen / Notationen:

M aktuell betrachtete Teilmenge von $\{x_1, ..., x_n\}$

M_{opt} bisher gefundene optimale Teilmenge von $\{x_1, ..., x_n\}$

```
Algorithmus OPTIMAL(M)
BEGIN
   IF (gewicht(M)<=y) THEN
      IF (gewicht(M)>gewicht(M_opt)) THEN
         M_opt := M;
      END(*IF*)
   END(*IF*)
END
```

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
   IF (i = n+1) THEN
         OPTIMAL(M);
   ELSE
         M := M ∪ {x_i};
         Rucksack1(i+1);
         M := M \ {x_i};
         Rucksack1(i+1);
   END(*IF*)
END
```

Erster Aufruf: **`Rucksack1(1)`** (mit **`M={ }`**)

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
   IF (i = n+1) THEN
         OPTIMAL(M);
   ELSE
         M := M ∪ {x_i};
         Rucksack1(i+1);
         M := M \ {x_i};
         Rucksack1(i+1);
   END(*IF*)
END
```

Rucksack1(i) untersucht Teilbäume mal mit x_i, mal ohne x_i

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
   IF (i = n+1) THEN
         OPTIMAL(M);
   ELSE
         M := M ∪ {x_i};
         Rucksack1(i+1);
         M := M \ {x_i};
         Rucksack1(i+1);
   END(*IF*)
END
```

Optimalität wird nur für Blätter geprüft

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
    IF (i = n+1) THEN
        OPTIMAL(M);
    ELSE
        M := M ∪ {x_i};
        Rucksack1(i+1);
        M := M \ {x_i};
        Rucksack1(i+1);
    END(*IF*)
END
```

Aufrufe (**n=3**):

Rucksack1(1)

Aktuelle Menge: **M={ }**

Geprüfte Mengen:

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
    IF (i = n+1) THEN
        OPTIMAL(M);
    ELSE
        M := M ∪ {x_i};
        Rucksack1(i+1);
        M := M \ {x_i};
        Rucksack1(i+1);
    END(*IF*)
END
```

Aufrufe (**n=3**):

Rucksack1(1)

Aktuelle Menge: **M={x_1}**

Geprüfte Mengen:

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
    IF (i = n+1) THEN
        OPTIMAL(M);
    ELSE
        M := M ∪ {x_i};
        Rucksack1(i+1);
        M := M \ {x_i};
        Rucksack1(i+1);
    END(*IF*)
END
```

Aufrufe (**n=3**):

Rucksack1(2)
Rucksack1(1)

Aktuelle Menge: $M=\{x_1\}$

Geprüfte Mengen:

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
    IF (i = n+1) THEN
        OPTIMAL(M);
    ELSE
        M := M ∪ {xi};
        Rucksack1(i+1);
        M := M \ {xi};
        Rucksack1(i+1);
    END(*IF*)
END
```

Aufrufe (**n=3**):

Rucksack1(2)
Rucksack1(1)

Aktuelle Menge: $M=\{x_1,x_2\}$

Geprüfte Mengen:

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
    IF (i = n+1) THEN
         OPTIMAL(M);
    ELSE
        M := M ∪ {x_i};
        Rucksack1(i+1);
        M := M \ {x_i};
        Rucksack1(i+1);
    END(*IF*)
END
```

Aufrufe (**n=3**):

Rucksack1(3)
Rucksack1(2)
Rucksack1(1)

Aktuelle Menge: $M=\{x_1,x_2\}$

Geprüfte Mengen:

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
    IF (i = n+1) THEN
         OPTIMAL(M);
    ELSE
        M := M ∪ {x_i};
        Rucksack1(i+1);
        M := M \ {x_i};
        Rucksack1(i+1);
    END(*IF*)
END
```

Aufrufe (**n=3**):

Rucksack1(3)
Rucksack1(2)
Rucksack1(1)

Aktuelle Menge: $M=\{x_1,x_2,x_3\}$

Geprüfte Mengen:

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
    IF (i = n+1) THEN
        OPTIMAL(M);
    ELSE
        M := M ∪ {x_i};
        Rucksack1(i+1);
        M := M \ {x_i};
        Rucksack1(i+1);
    END(*IF*)
END
```

Aufrufe (**n=3**):

Rucksack1(4)
Rucksack1(3)
Rucksack1(2)
Rucksack1(1)

Aktuelle Menge: $M=\{x_1,x_2,x_3\}$

Geprüfte Mengen:

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
    IF (i = n+1) THEN
        OPTIMAL(M);
    ELSE
        M := M ∪ {x_i};
        Rucksack1(i+1);
        M := M \ {x_i};
        Rucksack1(i+1);
    END(*IF*)
END
```

Aufrufe (**n=3**):

Rucksack1(4)
Rucksack1(3)
Rucksack1(2)
Rucksack1(1)

Aktuelle Menge: $M=\{x_1,x_2,x_3\}$

Geprüfte Mengen:

$\{x_1,x_2,x_3\}$

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
    IF (i = n+1) THEN
         OPTIMAL(M);
    ELSE
        M := M ∪ {x_i};
        Rucksack1(i+1);
        M := M \ {x_i};
        Rucksack1(i+1);
    END(*IF*)
END
```

Aufrufe (**n=3**):

Rucksack1(4)
Rucksack1(3)
Rucksack1(2)
Rucksack1(1)

Aktuelle Menge: $M=\{x_1, x_2, x_3\}$

Geprüfte Mengen:

$\{x_1, x_2, x_3\}$

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
    IF (i = n+1) THEN
        OPTIMAL(M);
    ELSE
        M := M ∪ {x_i};
        Rucksack1(i+1);
        M := M \ {x_i};
        Rucksack1(i+1);
    END(*IF*)
END
```

Aufrufe (**n=3**):

Rucksack1(3)
Rucksack1(2)
Rucksack1(1)

Aktuelle Menge: $M=\{x_1,x_2\}$

Geprüfte Mengen:

$\{x_1,x_2,x_3\}$

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
    IF (i = n+1) THEN
        OPTIMAL(M);
    ELSE
        M := M ∪ {x_i};
        Rucksack1(i+1);
        M := M \ {x_i};
        Rucksack1(i+1);
    END(*IF*)
END
```

Aufrufe (**n=3**):

Rucksack1(4)
Rucksack1(3)
Rucksack1(2)
Rucksack1(1)

Aktuelle Menge: **M=**$\{x_1, x_2\}$

Geprüfte Mengen:

$\{x_1, x_2, x_3\}$

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
    IF (i = n+1) THEN
        OPTIMAL(M);
    ELSE
        M := M ∪ {xi};
        Rucksack1(i+1);
        M := M \ {xi};
        Rucksack1(i+1);
    END(*IF*)
END
```

Aufrufe (**n=3**):

Rucksack1(4)
Rucksack1(3)
Rucksack1(2)
Rucksack1(1)

Aktuelle Menge: $M=\{x_1, x_2\}$

Geprüfte Mengen:

$\{x_1, x_2\}$

$\{x_1, x_2, x_3\}$

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
    IF (i = n+1) THEN
        OPTIMAL(M);
    ELSE
        M := M ∪ {x_i};
        Rucksack1(i+1);
        M := M \ {x_i};
        Rucksack1(i+1);
    END(*IF*)
END
```

Aufrufe (**n=3**):

Rucksack1(4)
Rucksack1(3)
Rucksack1(2)
Rucksack1(1)

Aktuelle Menge: $M=\{x_1,x_2\}$

Geprüfte Mengen:

$\{x_1,x_2\}$

$\{x_1,x_2,x_3\}$

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
    IF (i = n+1) THEN
         OPTIMAL(M);
    ELSE
         M := M ∪ {x_i};
         Rucksack1(i+1);
         M := M \ {x_i};
         Rucksack1(i+1);
    END(*IF*)
END
```

Aufrufe (**n=3**):

Rucksack1(3)
Rucksack1(2)
Rucksack1(1)

Aktuelle Menge: $M=\{x_1, x_2\}$

Geprüfte Mengen:

$\{x_1, x_2\}$
$\{x_1, x_2, x_3\}$

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
    IF (i = n+1) THEN
         OPTIMAL(M);
    ELSE
        M := M ∪ {x_i};
        Rucksack1(i+1);
        M := M \ {x_i};
        Rucksack1(i+1);
    END(*IF*)
END
```

Aufrufe (**n=3**):

Rucksack1(2)
Rucksack1(1)

Aktuelle Menge: **M=**$\{x_1\}$

Geprüfte Mengen:

$\{x_1, x_2\}$
$\{x_1, x_2, x_3\}$

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
    IF (i = n+1) THEN
         OPTIMAL(M);
    ELSE
         M := M ∪ {x_i};
         Rucksack1(i+1);
         M := M \ {x_i};
         Rucksack1(i+1);
    END(*IF*)
END
```

Aufrufe (**n=3**):

Rucksack1(3)
Rucksack1(2)
Rucksack1(1)

Aktuelle Menge: $M=\{x_1\}$

Geprüfte Mengen:

$\{x_1, x_2\}$
$\{x_1, x_2, x_3\}$

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
    IF (i = n+1) THEN
         OPTIMAL(M);
    ELSE
        M := M ∪ {x_i};
        Rucksack1(i+1);
        M := M \ {x_i};
        Rucksack1(i+1);
    END(*IF*)
END
```

Aufrufe (**n=3**):

Rucksack1(3)
Rucksack1(2)
Rucksack1(1)

Aktuelle Menge: $M=\{x_1,x_3\}$

Geprüfte Mengen:

$\{x_1,x_2\}$
$\{x_1,x_2,x_3\}$

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
    IF (i = n+1) THEN
         OPTIMAL(M);
    ELSE
        M := M ∪ {x_i};
        Rucksack1(i+1);
        M := M \ {x_i};
        Rucksack1(i+1);
    END(*IF*)
END
```

Aufrufe (**n=3**):

Rucksack1(4)
Rucksack1(3)
Rucksack1(2)
Rucksack1(1)

Aktuelle Menge: $M=\{x_1, x_3\}$

Geprüfte Mengen:

$\{x_1, x_2\}$
$\{x_1, x_2, x_3\}$

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
    IF (i = n+1) THEN
        OPTIMAL(M);
    ELSE
        M := M ∪ {xi};
        Rucksack1(i+1);
        M := M \ {xi};
        Rucksack1(i+1);
    END(*IF*)
END
```

Aufrufe (**n=3**):

Rucksack1(4)
Rucksack1(3)
Rucksack1(2)
Rucksack1(1)

Aktuelle Menge: $M=\{x_1,x_3\}$

Geprüfte Mengen:

$\{x_1,x_3\}$
$\{x_1,x_2\}$
$\{x_1,x_2,x_3\}$

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
    IF (i = n+1) THEN
         OPTIMAL(M);
    ELSE
        M := M ∪ {xi};
        Rucksack1(i+1);
        M := M \ {xi};
        Rucksack1(i+1);
    END(*IF*)
END
```

Aufrufe (**n=3**):

Rucksack1(4)
Rucksack1(3)
Rucksack1(2)
Rucksack1(1)

Aktuelle Menge: $M=\{x_1, x_3\}$

Geprüfte Mengen:

$\{x_1, x_3\}$
$\{x_1, x_2\}$
$\{x_1, x_2, x_3\}$

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
    IF (i = n+1) THEN
        OPTIMAL(M);
    ELSE
        M := M ∪ {x_i};
        Rucksack1(i+1);
        M := M \ {x_i};
        Rucksack1(i+1);
    END(*IF*)
END
```

Aufrufe (**n=3**):

Rucksack1(3)
Rucksack1(2)
Rucksack1(1)

Aktuelle Menge: $M=\{x_1\}$

Geprüfte Mengen:

$\{x_1, x_3\}$
$\{x_1, x_2\}$
$\{x_1, x_2, x_3\}$

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
    IF (i = n+1) THEN
        OPTIMAL(M);
    ELSE
        M := M ∪ {x_i};
        Rucksack1(i+1);
        M := M \ {x_i};
        Rucksack1(i+1);
    END(*IF*)
END
```

Aufrufe (**n=3**):

Rucksack1(4)
Rucksack1(3)
Rucksack1(2)
Rucksack1(1)

Aktuelle Menge: $M=\{x_1\}$

Geprüfte Mengen:

$\{x_1, x_3\}$
$\{x_1, x_2\}$
$\{x_1, x_2, x_3\}$

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
    IF (i = n+1) THEN
       OPTIMAL(M);
    ELSE
        M := M ∪ {x_i};
        Rucksack1(i+1);
        M := M \ {x_i};
        Rucksack1(i+1);
    END(*IF*)
END
```

Aufrufe (**n=3**):

Rucksack1(4)
Rucksack1(3)
Rucksack1(2)
Rucksack1(1)

Aktuelle Menge: $M=\{x_1\}$

Geprüfte Mengen:

$\{x_1\}$
$\{x_1, x_3\}$
$\{x_1, x_2\}$
$\{x_1, x_2, x_3\}$

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
    IF (i = n+1) THEN
        OPTIMAL(M);
    ELSE
        M := M ∪ {x_i};
        Rucksack1(i+1);
        M := M \ {x_i};
        Rucksack1(i+1);
    END(*IF*)
END
```

Aufrufe (**n=3**):

Rucksack1(4)
Rucksack1(3)
Rucksack1(2)
Rucksack1(1)

Aktuelle Menge: $M=\{x_1\}$

Geprüfte Mengen:

$\{x_1\}$

$\{x_1, x_3\}$

$\{x_1, x_2\}$

$\{x_1, x_2, x_3\}$

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
    IF (i = n+1) THEN
        OPTIMAL(M);
    ELSE
        M := M ∪ {x_i};
        Rucksack1(i+1);
        M := M \ {x_i};
        Rucksack1(i+1);
    END(*IF*)
END
```

Aufrufe (**n=3**):

Rucksack1(3)
Rucksack1(2)
Rucksack1(1)

Aktuelle Menge: $M=\{x_1\}$

Geprüfte Mengen:

$\{x_1\}$
$\{x_1, x_3\}$
$\{x_1, x_2\}$
$\{x_1, x_2, x_3\}$

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
    IF (i = n+1) THEN
        OPTIMAL(M);
    ELSE
        M := M ∪ {x_i};
        Rucksack1(i+1);
        M := M \ {x_i};
        Rucksack1(i+1);
    END(*IF*)
END
```

Aufrufe (**n=3**):

Rucksack1(2)
Rucksack1(1)

Aktuelle Menge: $M=\{x_1\}$

Geprüfte Mengen:

$\{x_1\}$
$\{x_1, x_3\}$
$\{x_1, x_2\}$
$\{x_1, x_2, x_3\}$

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
    IF (i = n+1) THEN
        OPTIMAL(M);
    ELSE
        M := M ∪ {x_i};
        Rucksack1(i+1);
        M := M \ {x_i};
        Rucksack1(i+1);
    END(*IF*)
END
```

Aufrufe (**n=3**):

Rucksack1(1)

Aktuelle Menge: **M={}**

Geprüfte Mengen:

$\{x_1\}$

$\{x_1, x_3\}$

$\{x_1, x_2\}$

$\{x_1, x_2, x_3\}$

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack1(i);
BEGIN
    IF (i = n+1) THEN
        OPTIMAL(M);
    ELSE
        M := M ∪ {x_i};
        Rucksack1(i+1);
        M := M \ {x_i};
        Rucksack1(i+1);
    END(*IF*)
END
```

Aufrufe (**n=3**):

Rucksack1(1)

Aktuelle Menge: **M={}**

Geprüfte Mengen:

$\{x_1\}$

$\{x_1, x_3\}$

$\{x_1, x_2\}$

$\{x_1, x_2, x_3\}$

Jetzt wie eben für **{}** statt für $\{x_1\}$: Mengen $\{\}, \{x_3\}, \{x_2\}, \{x_2, x_3\}$ werden durchlaufen.

Backtracking - Rucksackproblem

Vor dem ersten Aufruf des Algorithmus:

Einlesen von $\mathbf{g(x_1), ..., g(x_n)}$ und $\mathbf{y}$

Initialisieren von $\mathbf{M}$ und $\mathbf{M_{opt}}$ mit der leeren Menge.

Backtracking - Rucksackproblem

Erster Aufruf mit **`Rucksack1(1)`**:

Der Algorithmus prüft

- alle 2^n maximalen Pfade des Entscheidungsbaumes

- wenn er an einem Blatt angekommen ist, ob die dort stehende Menge M „besser" ist als das aktuelle Optimum M_{Opt}

□

Backtracking - Rucksackproblem

<u>Verbesserung: Abschneiden "sinnloser" Teilbäume (branch-and-bound-Technik):</u>

An jedem Knoten prüfen, ob geforderte Bedingungen überhaupt erfüllt werden können!

Backtracking - Rucksackproblem

```
ALGORITHMUS Rucksack2(i);
BEGIN
   IF (i == n+1) THEN
        OPTIMAL(M);
   ELSE
         M = M ∪ {xi};
         IF (gewicht(M) <= y) THEN
            Rucksack1(i+1);
         END(*IF*)
         M = M \ {xi};
         IF (gewicht(M) <= y) THEN
            Rucksack1(i+1);
         END(*IF*)
   END(*IF*)
END
```

Backtracking - Rucksackproblem

<u>Zeitkomplexität:</u>

$\mathbf{T_{max}(Rucksack1;n), T_{max}(Rucksack2;n) \in O(2^n)}$ □

`Rucksack2` ist effizienter als **`Rucksack1`**
(abhängig von der Wahl von $\mathbf{y}$ und der Menge $\mathbf{\{x_1, ..., x_n\}}$)

<u>Bemerkungen:</u>

Backtracking-Algorithmen sind im allgemeinen von exponentieller Zeitkomplexität.

Denkbar sind auch Entscheidungsbäume mit mehr als 2 Nachfolgern in den inneren Knoten (TSP, 8-Damen-Problem)

Backtracking - Rucksackproblem

`Rucksack1` in C implementieren (**`n=4`**):

Mit welcher Datenstruktur wollen wir eine Menge $\mathbf{M} \subseteq \{\mathbf{x}_1, \ldots, \mathbf{x}_4\}$ darstellen?

Backtracking - Rucksackproblem

`Rucksack1` in C implementieren (**`n=4`**):

Mit welcher Datenstruktur wollen wir eine Menge
$\mathbf{M} \subseteq \{\mathbf{x}_1, \dots, \mathbf{x}_4\}$ darstellen?

Bitfelder der Länge **`n`**:
`int m[4]`
Menge $\{\mathbf{x}_1, \mathbf{x}_2\}$ entspricht **`m[0]=1,m[1]=1,m[2]=0,m[3]=0`**

Was bedeutet dann $\mathbf{M} := \mathbf{M} \cup \{\mathbf{x}_3\}$?
Oder $\mathbf{M} := \mathbf{M} \setminus \{\mathbf{x}_1\}$?
Oder $\mathbf{M}_{opt} := \mathbf{M}$?

Backtracking - Rucksackproblem

`Rucksack1` in C implementieren (**`n=4`**):

Mit welcher Datenstruktur wollen wir eine Menge $\mathbf{M} \subseteq \{\mathbf{x}_1, \ldots, \mathbf{x}_4\}$ darstellen?

Bitfelder der Länge **`n`**:

`int m[4]`

Menge $\{\mathbf{x}_1, \mathbf{x}_2\}$ entspricht **`m[0]=1,m[1]=1,m[2]=0,m[3]=0`**

Was bedeutet dann $\mathbf{M} := \mathbf{M} \cup \{\mathbf{x}_3\}$? **`m[2]:=1;`**

Oder $\mathbf{M} := \mathbf{M} \setminus \{\mathbf{x}_1\}$? **`m[0]:=0;`**

Oder $\mathbf{M}_{opt} := \mathbf{M}$? **`for(i=0;i<n;++i){opt[i]:=m[i];}`**

Backtracking - Rucksackproblem

`Rucksack1` in C implementieren (**`n=4`**):

Mit welcher Datenstruktur wollen wir die Gegenstände darstellen?

Backtracking - Rucksackproblem

`Rucksack1` in C implementieren (**`n=4`**):

Mit welcher Datenstruktur wollen wir die Gegenstände darstellen?

```
typedef struct item{
    int id;
    int gewicht;
} Item;
```

Backtracking - Rucksackproblem

`Rucksack1` in C implementieren (**`n=4`**):

Gesamtgewicht einer Teilmenge **`M`** berechnen:

Backtracking - Rucksackproblem

`Rucksack1` in C implementieren (**`n=4`**):

Gesamtgewicht einer Teilmenge **`M`** berechnen:

```
int gewicht(int m[], Item items[], int n)
{
    int i, s=0;
    for(i=0; i<n; ++i){
        s = s + m[i]*items[i].gewicht;
    }
    return s;
}
```

Backtracking - Rucksackproblem

`Rucksack1` in C implementieren (**`n=4`**):

Menge auf Optimalität überprüfen:

Backtracking - Rucksackproblem

`Rucksack1` in C implementieren (**`n=4`**):

Menge auf Optimalität überprüfen:

```
void testopt(int m[],Item items[],int opt[],int y,int n)
{
    if ((gewicht(m, items, n) <= y)&&
        (gewicht(m, items, n) > gewicht(opt, items, n))){
        int i;
        for(i=0; i<n; ++i){
            opt[i] = m[i];
        }
    }
}
```

Backtracking - Rucksackproblem

`Rucksack1` in C implementieren (**`n=4`**):

```
void rucksack1(int m[],int opt[],Item items[],int
schranke,int ebene,int n)
{
    if (ebene == n){
        testopt(m,items,opt,schranke,n);
    }
    else {
        m[ebene] = 1;
        rucksack1(m,opt,items,schranke,ebene + 1,n);
        m[ebene] = 0;
        rucksack1(m,opt,items,schranke,ebene + 1,n);
    }
}
```

Backtracking - Rucksackproblem

`Rucksack1` in C implementieren (**`n=4`**):

```
int main()
{
    int n = 4;
    Item items[4];
    int m[] = {0,0,0,0}; //M = { }
    int opt[] = {0,0,0,0}; //Mopt = { }
    int y = 50; //obere Schranke
    int i;
    srand(time(NULL)); //Zufalls-Gewichte
    for (i=0; i<n; ++i){
        items[i].id = i+1;
        items[i].gewicht = rand()%100;
    }
   rucksack1(m,opt,items,y,0,n);//1. Aufruf für Ebene 0
}
```

Backtracking - Rucksackproblem

Bemerkungen:

Es war nicht nötig, einen Baum aufzubauen, also sich alle Knoten zu merken

Man konnte alte zulässige Lösungen getrost überschreiben

Dies wurde uns durch die Rekursion ermöglicht

Backtracking - TSP

- Wie lässt sich die Menge der zulässigen Lösungen als Baum darstellen?
- Wie kann man diesen Baum rekursiv durchlaufen (ohne alle Knoten zu speichern)?
- Mit welcher Datenstruktur stellen wird eine Rundreise dar? Wie speichern wir die Distanzen ziwschen den Städten?
- Wie berechnen wir die Länge einer Rundreise?
- Was könnte ein Kriterium sein für das Abschneiden von Ästen des Baums?

Backtracking - TSP

Ebene 0: Noch keiner Stadt ist eine Position in der Rundreise zugewiesen, d.h. jede Position hat den Wert **`-1`**:

`(-1,-1,-1,-1)`

Backtracking - TSP

Ebene 0: Noch keiner Stadt ist eine Position in der Rundreise zugewiesen, d.h. jede Position hat den Wert **`-1`**:

`(-1,-1,-1,-1)`

Ebene 1: Für die erste Stadt (ID **`0`**) wird jede Position probiert

`(0,-1,-1,-1)` **`(-1,0,-1,-1)`** **`(-1,-1,0,-1)`**

`(-1,-1,-1,0)`

Backtracking - TSP

Ebene 0: Noch keiner Stadt ist eine Position in der Rundreise zugewiesen, d.h. jede Position hat den Wert **`-1`**:

`(-1,-1,-1,-1)`

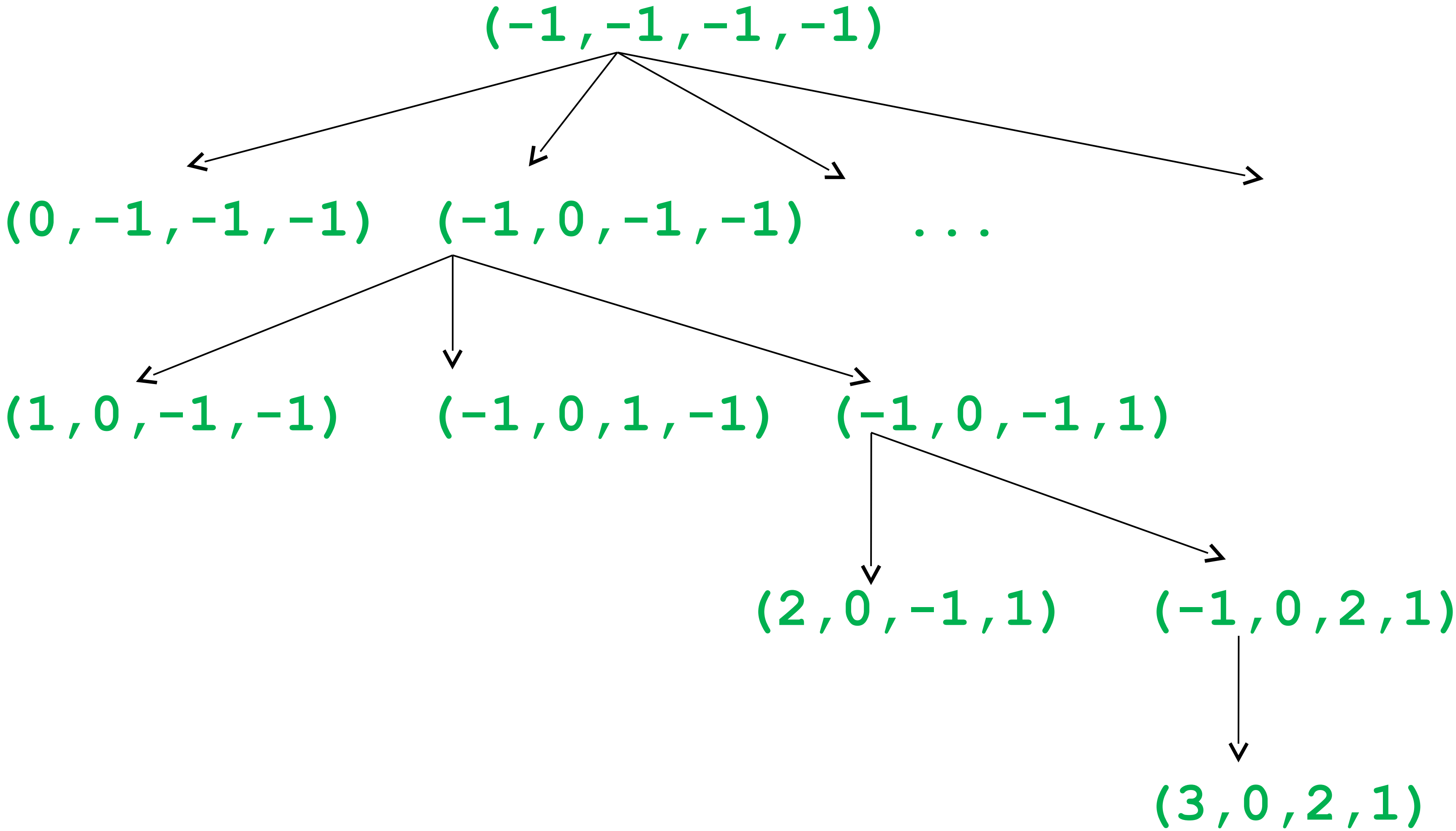
Ebene 1: Für die erste Stadt (ID **`0`**) wird jede Position probiert:

`(0,-1,-1,-1)` `(-1,0,-1,-1)` `(-1,-1,0,-1)`
`(-1,-1,-1,0)`

Ebene 2: Für jede Position der ersten Stadt (ID **`0`**) wird jede noch freie Position für die zweite Stadt (ID **`1`**) probiert:

`(0,1,-1,-1)` `(0,-1,1,-1)` `(0,-1,-1,1)`
`(1,0,-1,-1)` `(-1,0,1,-1)` `(-1,0,-1,1)`
`(1,-1,0,-1)` `(-1,1,0,-1)` `(-1,-1,0,1)`
`(1,-1,-1,0)` `(-1,1,-1,0)` `(-1,-1,1,0)`

Backtracking - TSP



Backtracking - TSP

- Wie lässt sich die Menge der zulässigen Lösungen als Baum darstellen?
- Wie kann man diesen Baum rekursiv durchlaufen (ohne alle Knoten zu speichern)?
- Mit welcher Datenstruktur stellen wird eine Rundreise dar? Wie speichern wir die Distanzen ziwschen den Städten?
- Wie berechnen wir die Länge einer Rundreise?
- Was könnte ein Kriterium sein für das Abschneiden von Ästen des Baums?

TSP

TSP – Pseudocode

```
ALGORITHMUS tsp(nummer)  //1.Aufruf: tsp(0)
BEGIN
   IF (nummer > n) THEN
      PrüfeOptimalität(R);
   ELSE
      FOR („Position pos in R noch nicht besetzt“) DO
         „Stadt nummer an Position pos in R setzen“;
         tsp(nummer+1);
         „Stadt nummer von Position pos in R wegnehmen“;
      END(*FOR*)
   END(*IF*)
END
```

TSP

- Wie lässt sich die Menge der zulässigen Lösungen als Baum darstellen?
- Wie kann man diesen Baum rekursiv durchlaufen (ohne alle Knoten zu speichern)?
- Mit welcher Datenstruktur stellen wird eine Rundreise dar? Wie speichern wir die Distanzen zwischen den Städten?
- Wie berechnen wir die Länge einer Rundreise?
- Was könnte ein Kriterium sein für das Abschneiden von Ästen des Baums?

TSP

<u>Wie speichern wir die Distanzen zwischen den Städten?</u>

```
int dist[4][4];

dist[m][k] == dist[k][m]
dist[m][m] == 0
```

TSP

<u>Mit welcher Datenstruktur stellen wird eine Rundreise dar?</u>

```
int perm[4];
```

Stadt **`k`** an Position **`m`** setzen: `perm[m]=k;`

Stadt **`k`** von Position **`m`** wegnehmen: `perm[m]=-1;`

Entfernung zwischen **`m`**-ter und **`(m+1)`**–ter Stadt in der Rundreise:
`dist[perm[m]][perm[m+1]];`

Auf Optimalität testen: ähnlich wie bei Rucksack-Problem

TSP

- Wie lässt sich die Menge der zulässigen Lösungen als Baum darstellen?
- Wie kann man diesen Baum rekursiv durchlaufen (ohne alle Knoten zu speichern)?
- Mit welcher Datenstruktur stellen wird eine Rundreise dar? Wie speichern wir die Distanzen ziwschen den Städten?
- Wie berechnen wir die Länge einer Rundreise?
- Was könnte ein Kriterium sein für das Abschneiden von Ästen des Baums?

TSP

Wie berechnen wir die Länge einer Rundreise?

```
perm[0] = 2;
perm[1] = 0;
perm[2] = 1;
perm[3] = 3;

dist[2][0] + dist[0][1] + dist[1][3]
+ dist[3][2]

for (i = 0; i<4; ++i){
     s = s + dist[perm[i]][perm[(i+1)%4]];
}
```

TSP

- Wie lässt sich die Menge der zulässigen Lösungen als Baum darstellen?
- Wie kann man diesen Baum rekursiv durchlaufen (ohne alle Knoten zu speichern)?
- Mit welcher Datenstruktur stellen wird eine Rundreise dar? Wie speichern wir die Distanzen ziwschen den Städten?
- Wie berechnen wir die Länge einer Rundreise?
- Was könnte ein Kriterium sein für das Abschneiden von Ästen des Baums?

TSP

TSP – Pseudocode mit branch and bound:

```
ALGORITHMUS tsp(nummer) //1.Aufruf: tsp(0)
BEGIN
   IF (nummer > n) THEN
      PrüfeOptimalität(R);
   ELSE
      FOR („Position pos in R noch nicht besetzt“) DO
         „Stadt nummer an Position pos in R setzen“;
         IF (distmin(R) < dist(R_opt)) THEN
            tsp(nummer+1);
         END(*IF*)
         „Stadt nummer von Position pos in R wegnehmen“;
      END(*FOR*)
   END(*IF*)
END
```

TSP

Minimale Länge **`distanz_min(int perm[], int dist[], int len)`** einer noch unvollständigen Rundreise **`perm`** berechnen:

- Berechne die minimale Distanz zwischen zwei Städten mit **`int minimum(int dist[], int len)`**

- Steht für eine Position die Stadt noch nicht fest, so kann man zu den benachbarten Städten in R zumindest die minimale Distanz ansetzen.