HTWK Leipzig
Fachbereich IMN
LV Algorithmen und Komplexität
Sommersemester 2005
Datum: 22.07.05

LV Algorithmen und Komplexität

Lösung von TSP mit „Branch & Bound“

Matthias Jauernig
03IN/PI

Lösung des „Traveling Salesman Problem“ mit „Branch-and-Bound“

Inhaltsangabe:

1. Problem / Aufgabenstellung .... Seite 3

2. Algorithmische Idee, Branch-and-Bound .... Seite 4 ff.

3. Verdeutlichung anhand eines Beispiels .... Seite 8 ff.

4. Algorithmus-Eigenschaften .... Seite 14 ff.
   - 4.1. Traversierung des Entscheidungsbaumes .... Seite 14 ff.
   - 4.2. Korrekter Aufbau des Entscheidungsbaumes .... Seite 16 ff.
   - 4.3. Algorithmus-Invarianten .... Seite 19
   - 4.4. Bedingungen für den Algorithmus .... Seite 19
   - 4.5. Korrektheit des Algorithmus .... Seite 19
   - 4.6. Termination des Algorithmus .... Seite 20

5. Exkurs: Rechnen mit +∞ und -∞ .... Seite 20 ff.
   - 5.1. Mathematisches Rechnen mit +∞ und -∞ .... Seite 20 ff.
   - 5.2. Rechnen mit Unendlichkeitswerten in Java .... Seite 22

6. Praktische Umsetzung .... Seite 23 ff.

7. Komplexitäten .... Seite 32

8. Quellennachweis .... Seite 32

1. Problem / Aufgabenstellung

Es wird ist folgendes Problem zu lösen: gegeben ist eine Menge von Städten S mit der Kardinalität n. Nimmt man weiterhin an, dass zwischen zwei beliebigen Städten u, v $\in$ S (u $\neq$ v) eine direkte Verbindung besteht, so kann man die Städte mit ihren Verbindungen als einen vollständigen Graphen G=(V, E) mit der Knotenmenge V und der Kantenmenge E betrachten. Der Graph ist gerichtet, was bedeutet, dass zwischen 2 beliebigen Städten unterschiedliche Längen für Hin- und Rückreise entstehen können. Jede Kante (u,v) $\in$ E des Graphen ist gewichtet, wobei die Gewichte die Länge der Reise von Stadt u nach Stadt v angeben.
In dieser Konstellation kann der Graph direkt als n*n-Matrix gespeichert werden mit den Längen der Wege zwischen den Städte als Matrixelemente. Diese „Distanzmatrix“ sei wie folgt definiert:

`dist:{0,…,n-1}x{0,…,n-1}`$\rightarrow \mathbb{N}$

$$dist = \begin{pmatrix} dist_{0,0} & dist_{0,1} & \dots & dist_{0,n-1} \\ dist_{1,0} & dist_{1,1} & \dots & dist_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ dist_{n-1,0} & dist_{n-1,1} & \dots & dist_{n-1,n-1} \end{pmatrix}$$

Dabei gilt $dist_{i,i} = \infty$ ($0 \leq i \leq n-1$), d.h. die Hauptdiagonale ist mit ∞ besetzt, da keine Verbindung von einer Stadt zu sich selber existieren soll.
dist(i,j) mit $0 \leq i,j \leq n-1$ stellt eine Zugriffsfunktion auf die Distanzmatrix dar, liefert also das Element $dist_{i,j}$ als Distanz von Stadt i zu Stadt j zurück.

Mit diesem als Distanzmatrix gegebenem Graphen soll nun folgendes Problem gelöst werden:
Gesucht ist eine Rundreise minimaler Länge durch alle n Städte.
Das zu lösende Problem kann als *Suchproblem* charakterisiert werden, da wir eine der Rundreisen mit der Eigenschaft der minimalen Länge suchen. Zudem ist es ein *Minimierungsproblem*, da eine minimale Rundreise gesucht wird.
Anders ausgedrückt suchen wir einen Hamiltonkreis minimaler Länge im Graphen (einen, der alle Knoten einschließt). Wir suchen also einen minimalen Weg durch alle n Städte, wobei wir keine Stadt doppelt besuchen und am Ende zur Ausgangs-Stadt zurückkehren wollen.

Als Lösung des TSP erwarten wir die Distanzmatrix des Ausgangsproblems, in der für nicht zur Rundreise gehörige Kanten die Entfernung „∞“ eingetragen ist. Da ein Knoten genau einmal durchlaufen wird, existiert in der Lösungsmatrix damit stets genau ein Element pro Zeile und pro Spalte, welches $\neq \infty$ ist $\rightarrow$ dies kann als Permutation der n Städte angesehen werden und bildet somit die Rundreise durch alle Städte.

Rundreiseproblem mit Permutationen ausgedrückt: Gegeben ist eine Folge von Zahlen 0, 1, …, n-1, welche die n Städte repräsentieren. Die Distanzen zwischen den Städten seien als Distanzmatrix dist gegeben. Gesucht ist nun eine Permutation p_j ($1 \leq j \leq n!$) aus der Menge aller Permutationen P dieser n Zahlen, welche nur 1 Zyklus besitzt und folgendem Kriterium genügt:

$$length = \min_{1 \leq j \leq n!} \left[\left(\sum_{i=0}^{n-2} dist(p_j(i), p_j(i+1)) \right) + dist(p_j(n-1), p_j(0)) \right] \qquad (1)$$

2. Algorithmische Idee, Branch-and-Bound

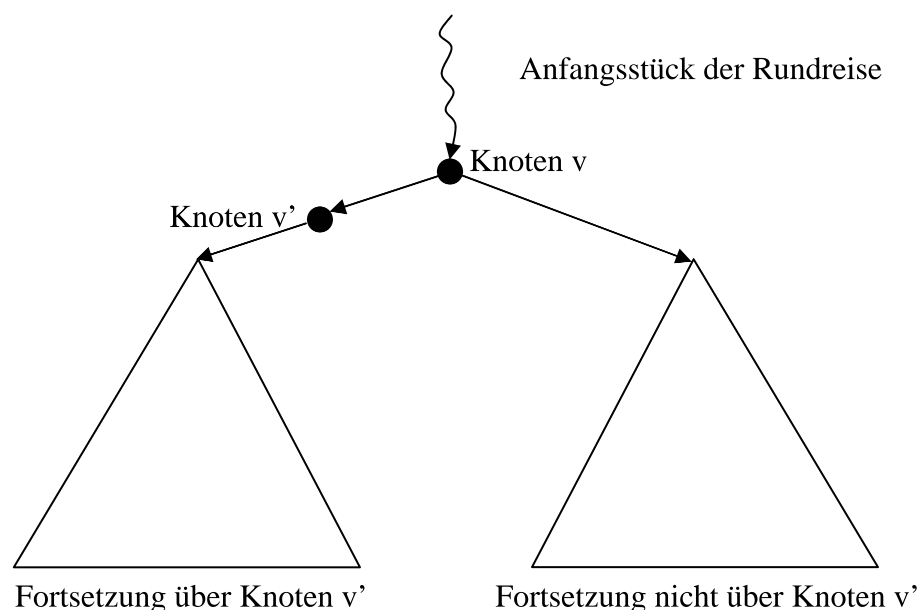
O.B.d.A. (in einem Kreis kann man an einer beliebigen Stelle beginnen) gehen wir davon aus, dass die Rundreise bei Stadt 0 beginnt und dort auch wieder endet (Knoten 0 im Graph).

Die triviale Lösung erhält man durch Aufzählung aller Permutationen der Knotenmenge und durch Anwenden von (1). Die Kardinalität der Menge aller Permutationen von n Zahlen beträgt n!. Unter der Annahme, dass die Rundreise bei Knoten 0 beginnen möge, sind zwar „nur noch“ (n-1)! Permutationen zu betrachten, da der Rechenaufwand allerdings trotzdem mit O(n!) anwächst, ist es selbst für relativ kleine Eingabegrößen wie z.B. n=1000 unvorstellbar, eine Lösung in vertretbarer Zeit zu finden.

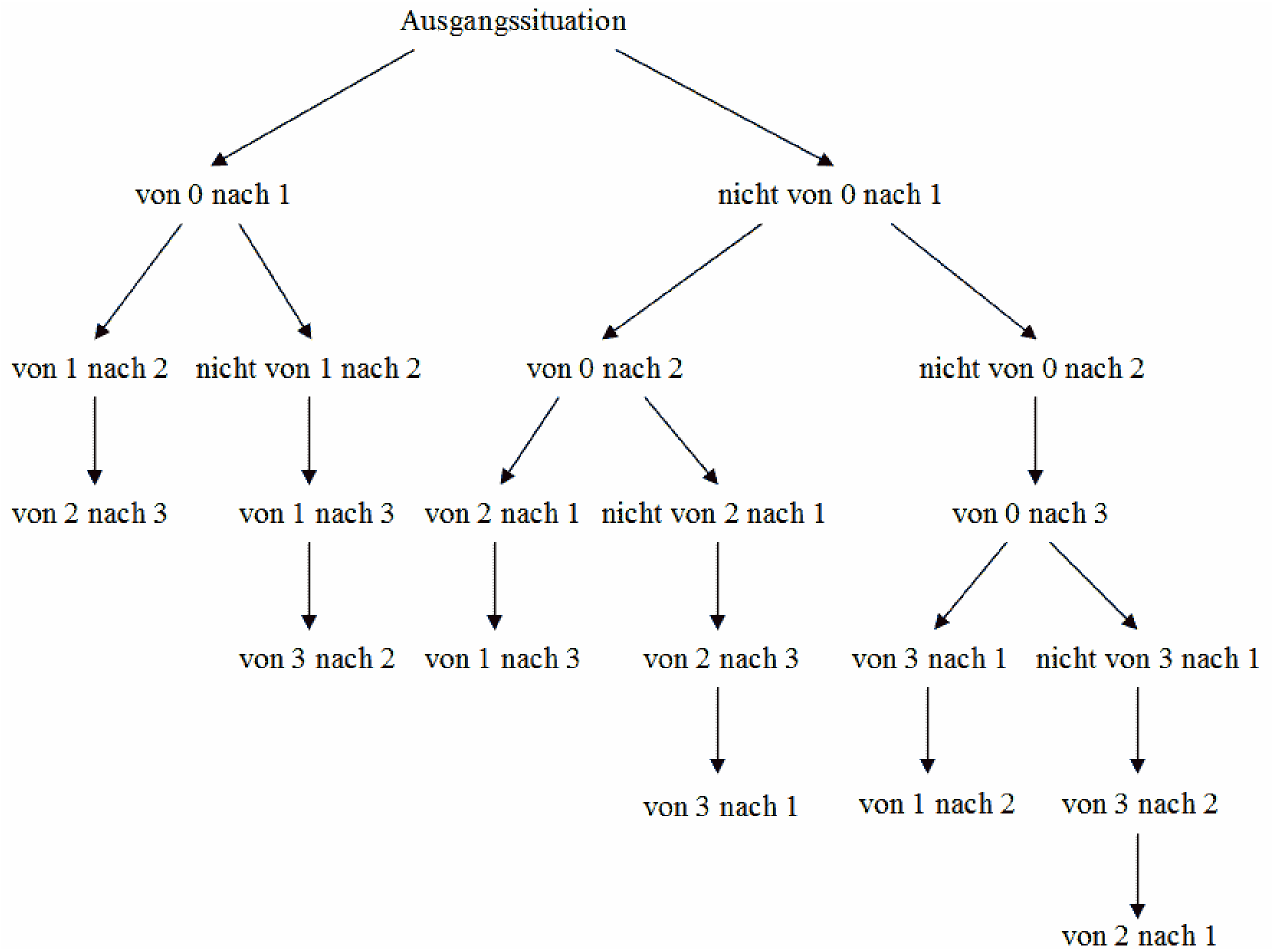
TSP ist ein NP-vollständiges Problem, d.h. es ist kein deterministischer Algorithmus bekannt, mit welchem eine Lösung in polynomieller Zeit gefunden werden könnte.
Und trotzdem wurden mit der Zeit Algorithmen entwickelt, die bei „normalen“ Eingabedaten eine Menge Rechenaufwand einsparen können, indem sie die Eingabedaten geschickt anordnen und Datenmengen von der Betrachtung komplett ausschließen. Auf einen dieser Algorithmen, den Branch-and-Bound-Algorithmus, möchte ich im Folgenden vertieft eingehen.

Allgemein ist Branch-and-Bound als eine Variante des Backtrackings anzusehen. Doch anders als beim Backtracking, wo eine erschöpfende Suche durchgeführt wird, um eine optimale Lösung zu finden, kann beim Branch-and-Bound-Algorithmus das Suchfeld eingeschränkt werden, indem Teilprobleme, die zu keiner optimalen Lösung führen können, vernachlässigt werden.

Der Algorithmus arbeitet in 2 Schritten. In der Annahme, dass bereits ein Anfangsstück einer Rundreise festgelegt ist und dass dieses Anfangsstück keine Rundreise durch den Graphen darstellt, wird im 1. Schritt der Suchraum zerlegt durch eine vollständige Fallunterscheidung über die Fortsetzung des Anfangsstücks der Rundreise.
Durch diese Zerlegung wird ein binärer Entscheidungsbaum aufgebaut, in welchem an jedem Knoten eine Entscheidung zwischen „Nimm Knoten x auf“ und „Nimm Knoten x nicht auf“ (mit $x \in V$) getroffen wird.
Allgemein lässt sich der Entscheidungsbaum somit wie folgt darstellen:



Der vollständige Entscheidungsbaum für n=4 Knoten lässt sich z.B. so darstellen:



Der Aufbau des Entscheidungsbaumes geschieht derart, dass bei jedem Baum-Knoten mit der dort gültigen Distanzmatrix (gültig heißt: mit dem aktuellen Anfangsstück der Rundreise konform) geprüft wird, welche Städte von der aktuellen Stadt i aus erreichbar sind.
Folgende Situationen sind dabei zu unterscheiden (eine genauere Betrachtung dieser findet in 4.2 statt):

a) Das aktuelle Anfangsstück der Rundreise hat die Länge n-1: in diesem Fall gehen wir von i zurück zum Knoten j=0 und komplettieren somit die Rundreise.
b) Für die Menge I der von i aus erreichbaren Städte gilt $|I\setminus\{0\}| > 1$: das bedeutet, dass mehr als eine Stadt von i aus erreichbar ist. Wir wählen die zahlenmäßig kleinste Stadt j aus I aus und führen eine Fallunterscheidung durch zwischen: A) „Setze Rundreise über j fort“, B) „Setze Rundreise nicht über j fort“. Für beide Entscheidungen sind neue Distanzmatrizen zu bilden.
   Bei A) geschieht dies dadurch, dass man I) alle in j eingehenden Kanten bis auf (i,j), II) alle aus i ausgehenden Kanten bis auf (i,j) sowie die Kante $(j,i) = \infty$ (also nicht gangbar) setzt, was eine Fortsetzung des aktuellen Anfangsstücks einer Rundreise über Knoten j bedeutet.
   Bei B) wird die Distanzmatrix so aktualisiert, dass die Kante $(i,j) = \infty$ gesetzt wird, was dem Ausschluss der Fortsetzung des aktuellen Anfangsstücks der Rundreise über Knoten j entspricht.
c) Für die Menge I der von i aus erreichbaren Städte gilt $|I\setminus\{0\}| = 1$: das bedeutet, dass uns keine Wahl bleibt – wir müssen das aktuelle Anfangsstück der Rundreise über den einzigen in I vorhandenen Knoten j fortsetzen. Die Distanzmatrix für den Teilbaum der Entscheidung wird gemäß 2.A) aktualisiert.

Bei n Knoten hat der Entscheidungsbaum für einen festgelegten Startknoten *(n-1)!* Blätter, ein Pfad von der Wurzel des Baumes zu einem Blatt (und zurück zum Startknoten) stellt dabei jeweils eine Rundreise dar.
Blätter sind auf Baumebenen $\geq n-1$ *und* $\leq (n-1)*n/2$ angeordnet.
Ein Entscheidungsbaum von n Knoten besitzt $K_n = (n-1)*(K_{n-1}+1)$ Knoten (mit n>1 und K_1=1).

Der Vorgang der Zerlegung wird als *Branching* („Verzweigen") bezeichnet.

Im 2. Schritt wird zunächst für jedes Teilproblem (an jedem inneren Knoten des Entscheidungsbaumes für den linken sowie rechten Sohn eines Knoten) eine **untere Schranke U** für die Kosten berechnet, welche mit dem aktuellen Anfangsstück einer Rundreise zu erwarten sind. U gibt an, wie hoch die Kosten für jede mögliche Lösung des Teilproblems mindestens sind:

Seien i ein Knoten im Graphen und A der zu durchsuchende Abschnitt des Lösungsraumes.
A stellt hier die Menge der in der betrachteten Situation begehbaren Kanten dar, $dist_A$ spiegelt den verbleibenden Teil des Lösungsraums wider (die Lösungen, welche mit dem aktuellen Anfangsstück einer Rundreise erreicht werden können).
I ist die Menge der mit einem Knoten i adjazenten Knoten.
Wenn j der Knoten vor i und k der Knoten nach i auf einer Rundreise ist, dann hat diese Rundreise durch den Graphen die Kosten:

$$K = \sum_{i=0}^{n-1} dist(i,k) = \frac{1}{2} * \sum_{i=0}^{n-1} \left(dist(i,k) + dist(j,i) \right)$$

Dann gilt für die untere Schranke:

$$U = \frac{1}{2} * \sum_{i=0}^{n-1} \left(\min_{j \in I \setminus \{i\}} dist_A(i,j) + \min_{j \in I \setminus \{i\}} dist_A(j,i) \right)$$

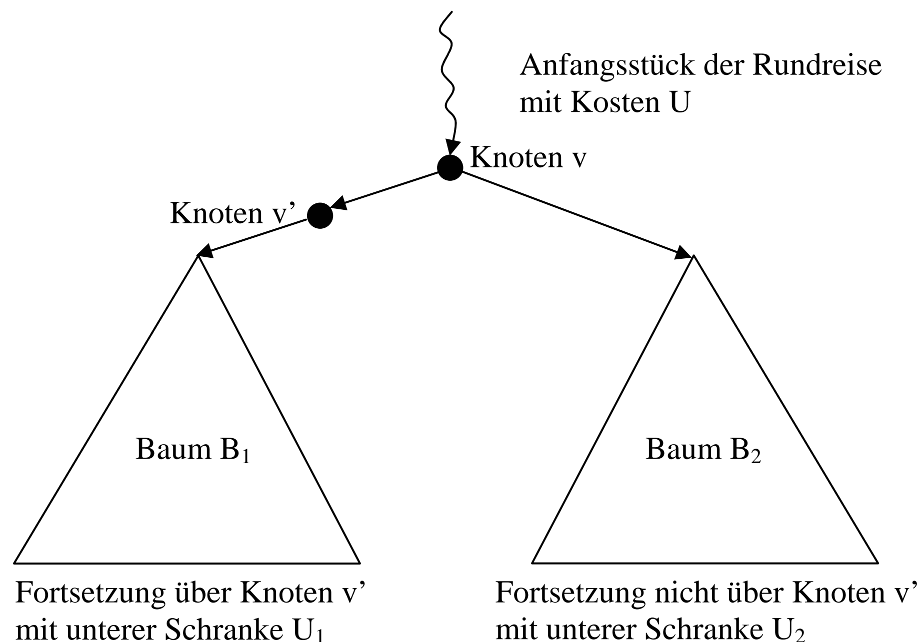
Begründung:

Alle Knoten müssen im verbleibenden Problem jeweils einmal besucht und wieder verlassen werden. Kleiner als die Summe der minimalen Kosten um in einen Knoten hinein und aus ihm wieder heraus zu kommen kann der Weg der kürzesten Rundreise dabei nicht werden. (Es gilt: $U \leq K$). Die Halbierung ist notwendig, da die Kosten zum Herausgehen aus einem Knoten i in einen Knoten j und Kosten zum Hereingehen aus einem Knoten k in den Knoten j doppelt gezählt werden, aber nur einfach betrachtet werden dürfen.

Zusätzlich zur unteren Schranke U wird eine **obere Schranke O** eingeführt, welche die minimalen Kosten der bisher gefundenen Rundreisen angibt. Wurde noch keine Rundreise gefunden, so ist O „∞" – erst im Lauf der Abarbeitung des Entscheidungsbaumes wird O durch die Kosten der bis zum jeweiligen Schritt gefundenen minimalen Rundreise ersetzt.
O gibt an, welche Teilprobleme weiter untersucht werden müssen. Gilt für eine untere Schranke U eines Teilproblems $O \leq U$, so kann über dieses Anfangsstück der Rundreise keine neue minimale Lösung gefunden werden, da die untere Schranke durch weitere Entscheidungen nur größer werden oder gleich bleiben kann, aber nicht kleiner wird. Der unter dem Teilproblem liegende Baum kann also „abgeschnitten" werden und ist nachfolgend nicht weiter zu betrachten.
Da somit einige Teilprobleme nicht weiter untersucht werden, wird der 2. Schritt des Algorithmus' als *Bounding* („Begrenzen") bezeichnet.

Beginnend mit dem Ursprungsproblem werden beide Schritte („Branch" und „Bound") solange wiederholt, bis alle Problemzweige eliminiert/durchlaufen wurden und eine optimale Lösung gefunden wurde.

An jedem Knoten des Baumes treffen wir eine Entscheidung über die Fortsetzung des Anfangsstücks der Rundreise. Folgende Abbildung stellt die Situation dar:



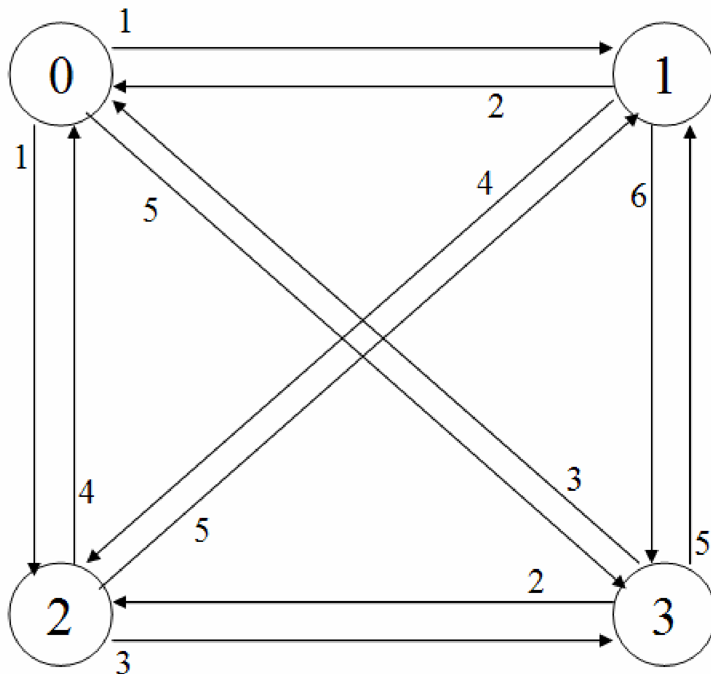
Es muss gelten: $U < O$, da wir das Anfangsstück der Rundreise nicht weiter verfolgen würden.
In dieser Situation sind 3 Fälle denkbar:

1. $U_1 \leq U_2$: Unter der Bedingung $U_1 < O$ verzweigen wir zunächst in den Teilbaum B_1. Nach Abarbeitung dessen verzweigen wir weiterhin in B_2, falls $U_2 < O$ gilt. Sollte $U_1 \geq O$ sein, so werden die Teilbäume B_1 und B_2 nicht weiter betrachtet.
2. $U_1 > U_2$: Unter der Bedingung $U_2 < O$ verzweigen wir zunächst in den Teilbaum B_2. Nach Abarbeitung dessen verzweigen wir weiterhin in B_1, falls $U_1 < O$ gilt. Sollte $U_2 \geq O$ sein, so werden die Teilbäume B_2 und B_1 nicht weiter betrachtet.
3. B_2 existiert nicht: Dies tritt auf, wenn eine Fortsetzung des Anfangsstücks einer Rundreise von Stadt i aus nur durch Hinzunahme eines Folgeknotens j zum Anfangsstück möglich ist, aber nicht durch Ausschluss dessen. In diesem Fall verzweigen wir unter der Bedingung $U_1 < O$ nur in den Teilbaum B_1. Sollte $U_1 \geq O$ sein, so wird der Teilbaum B_1 nicht weiter betrachtet.

Zu Beginn des Algorithmus' wird eine erste obere Schranke $\neq \infty$ dadurch gefunden, dass ein Weg durch den Entscheidungsbaum gemäß der Berechnung der kleinsten unteren Schranke an jedem Knoten für seine beiden Söhne durchlaufen wird, um eine erste Rundreise zu finden. Die obere Schranke O wird dann auf die Kosten dieser Rundreise gesetzt. Von da ausgehend können wir kürzere Rundreisen finden, indem wir die Teilbäume mit $U < O$ weiter betrachten und die Teilbäume mit $O \leq U$ abschneiden.

3. Verdeutlichung anhand eines Beispiels

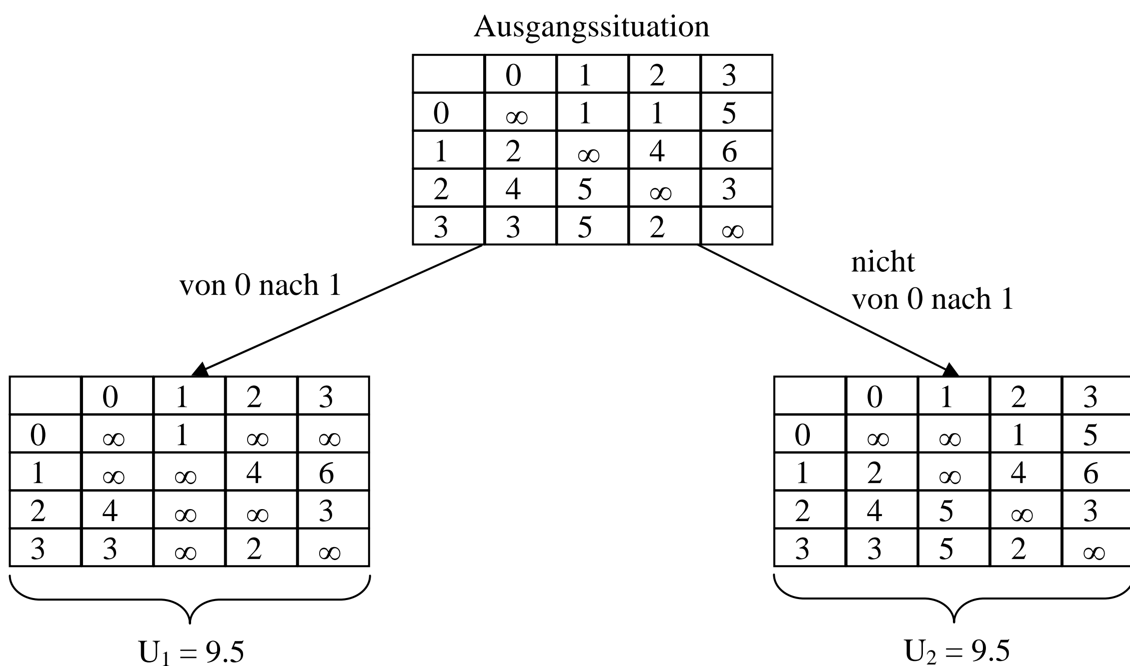
- Problem-Graph:



- Distanzmatrix des Ausgangsproblems:

	0	1	2	3
0	∞	1	1	5
1	2	∞	4	6
2	4	5	∞	3
3	3	5	2	∞

- Aufbau des Entscheidungsbaumes (dabei gilt am Anfang für die obere Schranke: O = ∞):



Ausgangssituation

	0	1	2	3
0	∞	1	1	5
1	2	∞	4	6
2	4	5	∞	3
3	3	5	2	∞

von 0 nach 1

	0	1	2	3
0	∞	1	∞	∞
1	∞	∞	4	6
2	4	∞	∞	3
3	3	∞	2	∞

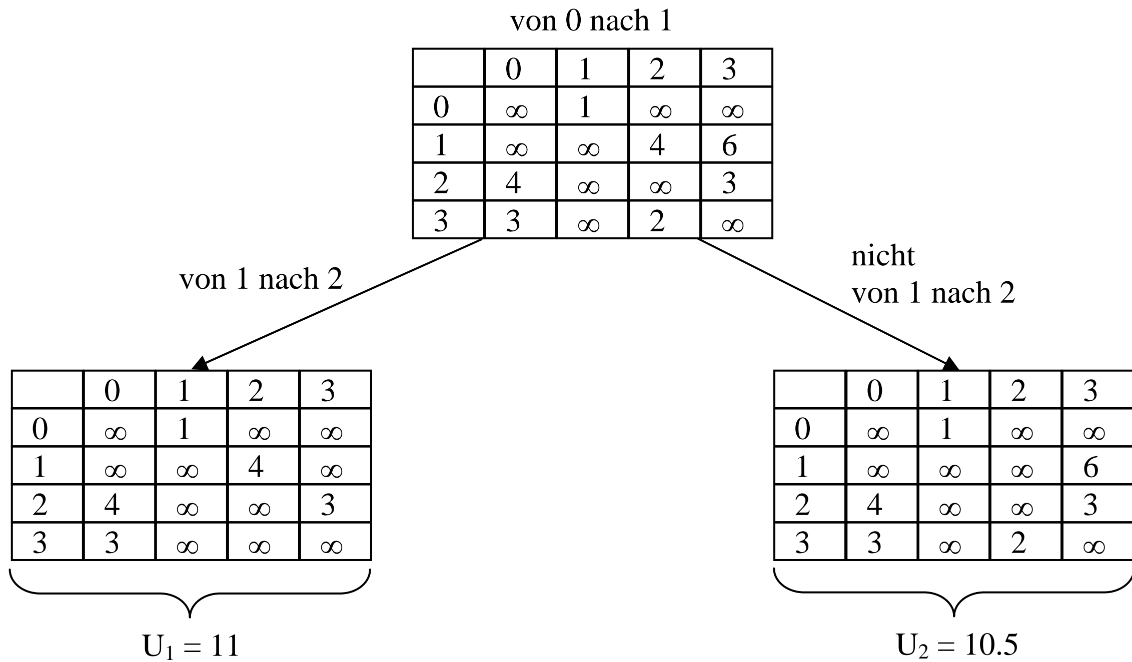
$U_1 = 9.5$

nicht von 0 nach 1

	0	1	2	3
0	∞	∞	1	5
1	2	∞	4	6
2	4	5	∞	3
3	3	5	2	∞

$U_2 = 9.5$

Die untere Schranke U für die beiden Teilprobleme ist identisch, weshalb wir zunächst den linken oder rechten Pfad weiter verfolgen könnten. Wir wollen hier zuerst den linken Pfad betrachten.



von 0 nach 1

	0	1	2	3
0	∞	1	∞	∞
1	∞	∞	4	6
2	4	∞	∞	3
3	3	∞	2	∞

von 1 nach 2

	0	1	2	3
0	∞	1	∞	∞
1	∞	∞	4	∞
2	4	∞	∞	3
3	3	∞	∞	∞

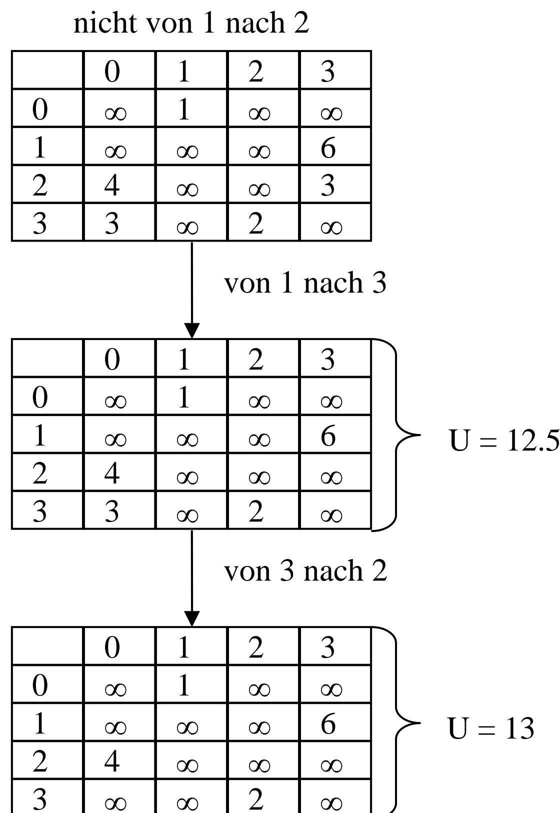
$U_1 = 11$

nicht von 1 nach 2

	0	1	2	3
0	∞	1	∞	∞
1	∞	∞	∞	6
2	4	∞	∞	3
3	3	∞	2	∞

$U_2 = 10.5$

Die untere Schranke U des rechten Sohnes ist kleiner als die untere Schranke des linken Sohnes, weshalb der Pfad über den rechten Sohn weiter verfolgt wird.



nicht von 1 nach 2

	0	1	2	3
0	∞	1	∞	∞
1	∞	∞	∞	6
2	4	∞	∞	3
3	3	∞	2	∞

von 1 nach 3

	0	1	2	3
0	∞	1	∞	∞
1	∞	∞	∞	6
2	4	∞	∞	∞
3	3	∞	2	∞

$U = 12.5$

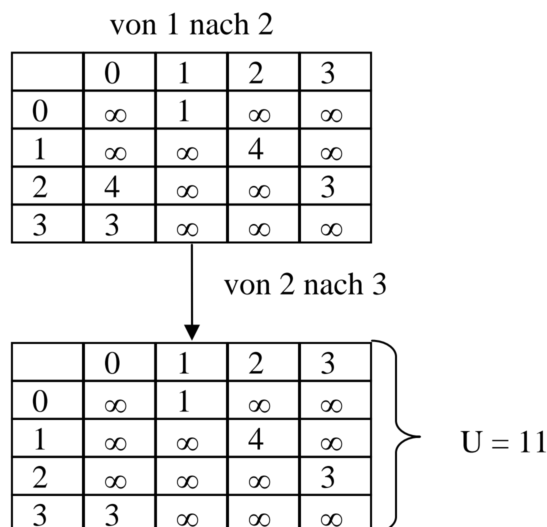
von 3 nach 2

	0	1	2	3
0	∞	1	∞	∞
1	∞	∞	∞	6
2	4	∞	∞	∞
3	∞	∞	2	∞

$U = 13$

In der Situation „nicht von 1 nach 2“ haben wir nur noch eine Option, von Knoten 1 weiter zu kommen und zwar über Knoten 3. Ebenso müssen wir dann vom Knoten 3 weiter zu Knoten 2 gehen, eine Entscheidung hinsichtlich „gehe nicht von 3 nach 2“ ist nicht möglich, da wir Knoten 1 schon besucht haben und noch nicht zu Knoten 0 zurückkehren dürfen, da wir dann Knoten 2 außer Acht gelassen hätten.
Nachdem wir „von 3 nach 2“ gegangen sind, ist die Rundreise beendet. Man kann sie auch an der Matrix ablesen: 0 $\rightarrow$ 1 $\rightarrow$ 3 $\rightarrow$ 2 und am Ende wieder zurück zu Knoten 0. Die untere Schranke in dieser finalen Situation entspricht der exakten Kosten der obigen Rundreise: U=13. Da wir eine Rundreise gefunden haben, bei der U < O gilt (denn offensichtlich gilt: U=13 < O=∞), wird die neue obere Schranke auf die bis dato kleinste Rundreise gesetzt, also gilt ab jetzt O=13.

Wir gehen im Entscheidungsbaum nun solange zurück, bis wir an eine Stelle kommen, an der wir noch einen anderen Weg einschlagen können, dies zugunsten einer kleineren unteren Schranke aber noch nicht getan haben.
Die erste solche Situation tritt beim Knoten mit der Entscheidung „von 0 nach 1“ auf. Zuerst hatten wir uns dafür entschieden „nicht von 1 nach 2“ zu gehen. Da der Pfad „von 1 nach 2“ mit den Kosten U=11 belegt ist und U < O (11 < 13) gilt, beschreiten wir diesen Weg:

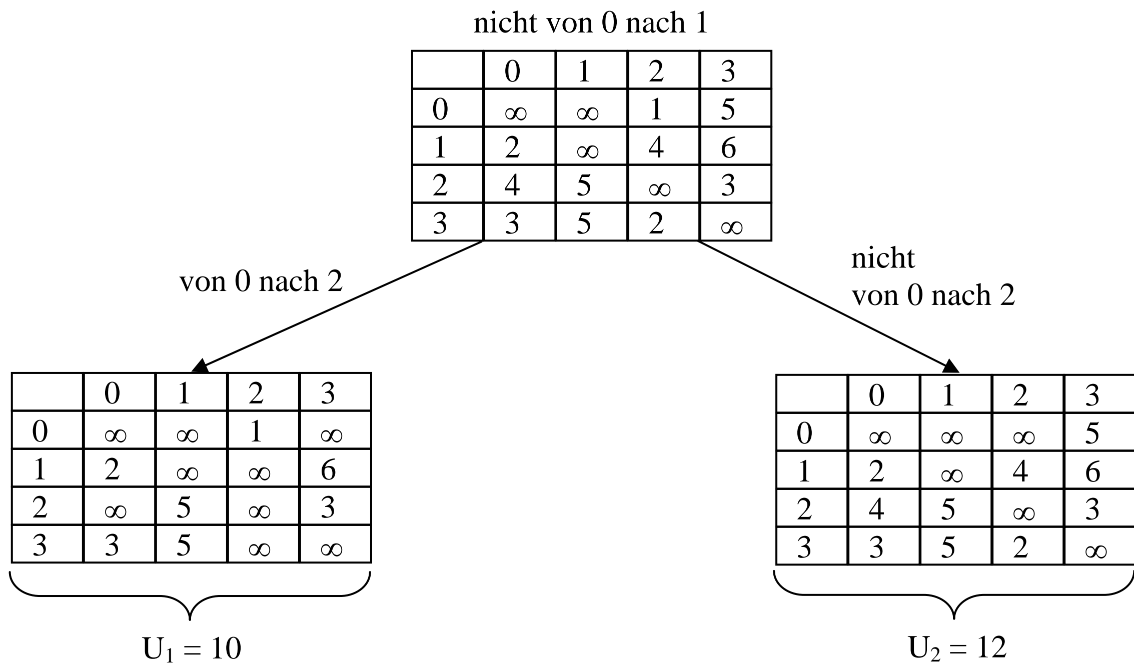


	0	1	2	3
0	∞	1	∞	∞
1	∞	∞	4	∞
2	4	∞	∞	3
3	3	∞	∞	∞

	0	1	2	3
0	∞	1	∞	∞
1	∞	∞	4	∞
2	∞	∞	∞	3
3	3	∞	∞	∞

In der Situation nach der Entscheidung „von 1 nach 2“ zu gehen, bleibt uns nichts anderes übrig, als „von 2 nach 3“ zu verzweigen. Zwar ist der Weg von 2 nach 0 noch frei, allerdings wäre über diesen die Rundreise nicht beendet.
Nachdem wir „von 2 nach 3“ gegangen sind, können wir weiter von 3 nach 0 laufen und haben wieder eine Rundreise gefunden: 0 $\rightarrow$ 1 $\rightarrow$ 2 $\rightarrow$ 3 und wieder zurück zu 0. Die exakten Kosten dieser Rundreise betragen U=11. Dies ist kleiner der bisherigen oberen Schranke O=13, also kann eine neue obere Schranke mit O=11 gesetzt werden und die neue bis dato kleinste Rundreise wurde gefunden.

Wie zuvor gehen wir im Entscheidungsbaum wieder solange zurück, bis wir uns für einen anderen Weg entscheiden können. Dabei kommen wir zur Ausgangssituation zurück, in der wir „von 0 nach 1“ gegangen sind. Nun werden wir den Weg „nicht von 0 nach 1“ weiter verfolgen, da dieser mit seiner unteren Schranke von U=9.5 ebenfalls kleiner der aktuellen oberen Schranke O=11 ist.



nicht von 0 nach 1

	0	1	2	3
0	∞	∞	1	5
1	2	∞	4	6
2	4	5	∞	3
3	3	5	2	∞

von 0 nach 2

	0	1	2	3
0	∞	∞	1	∞
1	2	∞	∞	6
2	∞	5	∞	3
3	3	5	∞	∞

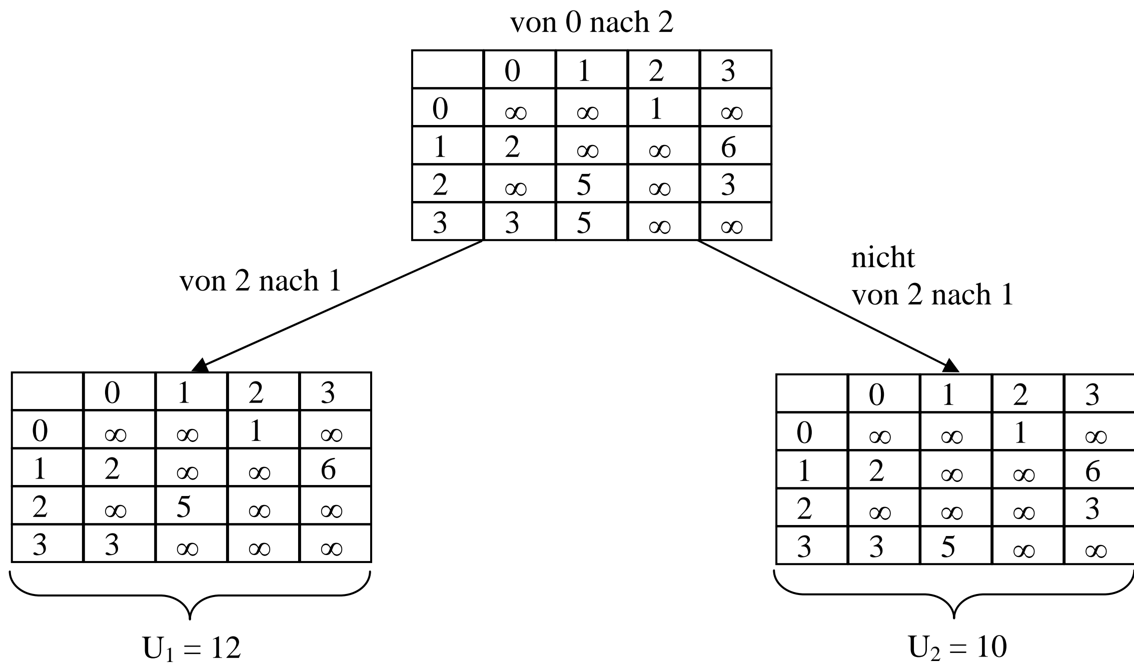
$U_1 = 10$

nicht von 0 nach 2

	0	1	2	3
0	∞	∞	∞	5
1	2	∞	4	6
2	4	5	∞	3
3	3	5	2	∞

$U_2 = 12$

Nach dem Prinzip der geringsten unteren Schranke beschreiten wir den Pfad „von 0 nach 2“:



von 0 nach 2

	0	1	2	3
0	∞	∞	1	∞
1	2	∞	∞	6
2	∞	5	∞	3
3	3	5	∞	∞

von 2 nach 1

	0	1	2	3
0	∞	∞	1	∞
1	2	∞	∞	6
2	∞	5	∞	∞
3	3	∞	∞	∞

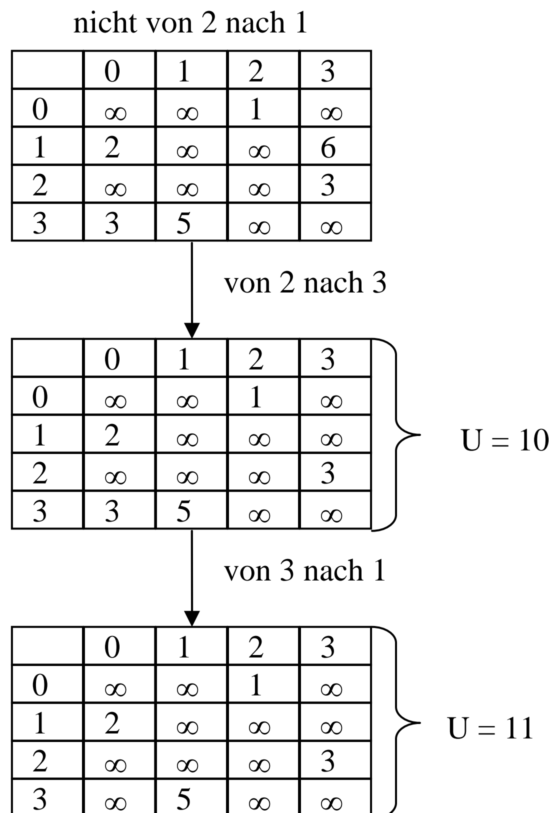
$U_1 = 12$

nicht von 2 nach 1

	0	1	2	3
0	∞	∞	1	∞
1	2	∞	∞	6
2	∞	∞	∞	3
3	3	5	∞	∞

$U_2 = 10$

Nach dem Beschreiten des Pfades „von 0 nach 2“ müssen wir vom Knoten 2 weiter gehen. Schaut man in Zeile 2 der Matrix in dieser Situation, so können wir vom Knoten 2 aus die Wege „von 2 nach 1“ oder „von 2 nach 3“ beschreiten – es ist also eine Fallunterscheidung zwischen „von 2 nach 1“ und „nicht von 2 nach 1“ möglich.
Hier entscheiden wir uns für den Pfad „nicht von 2 nach 1“, da dieser eine geringere untere Schranke besitzt (U_2=10 < U_1=12).



	0	1	2	3
0	∞	∞	1	∞
1	2	∞	∞	6
2	∞	∞	∞	3
3	3	5	∞	∞

	0	1	2	3
0	∞	∞	1	∞
1	2	∞	∞	∞
2	∞	∞	∞	3
3	3	5	∞	∞

	0	1	2	3
0	∞	∞	1	∞
1	2	∞	∞	∞
2	∞	∞	∞	3
3	∞	5	∞	∞

In der Situation, in der wir uns entschieden haben, „nicht von 2 nach 1" zu gehen, müssen wir uns für den Weg „von 2 nach 3" entscheiden, da dieser als einzige Alternative zur Verfügung steht.
Ebenso müssen wir anschließend „von 3 nach 1" gehen – zwar ist der Weg „von 3 nach 0" ebenfalls möglich, würde aber keine vollendete Rundreise erzeugen.

Nachdem wir „von 3 nach 1" gegangen sind, ist eine neue Rundreise vollendet: 0 → 2 → 3 → 1 und zurück zu 0. Diese Rundreise besitzt die Kosten U=11 – da diese gleich der oberen Schranke sind (U=O=11), wurde keine neue kürzeste Rundreise gefunden, die alte kürzeste Rundreise bleibt also bestehen.

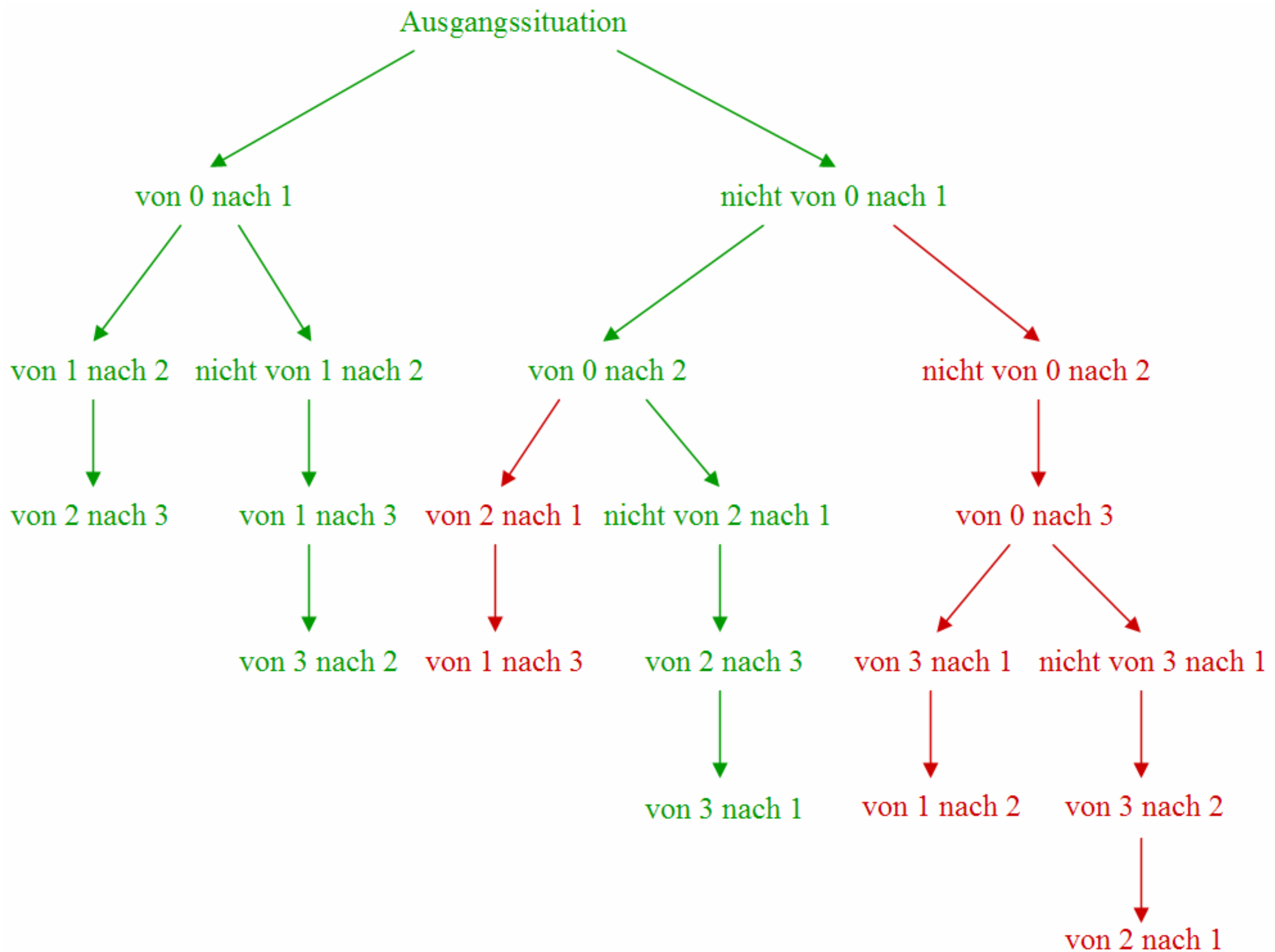
Wie in den Fällen zuvor auch durchlaufen wir den Baum wieder rückwärts in Richtung Wurzel. In der Situation, nach der wir „von 0 nach 2" gegangen sind, könnten wir uns nun dafür entscheiden statt „nicht von 2 nach 1" eben doch „von 2 nach 1" zu gehen.
Allerdings liegt die untere Schranke dieses Teilbaumes bei U=12.5 und damit über der aktuellen oberen Schranke von O=11. Wir sind in einer Situation, in der wir den kompletten Teilbaum abschneiden können. Weil: die Äste in diesem Teilbaum können uns nur zu Situationen führen, in denen die untere Schranke größer oder gleich der unteren Schranke der Teilbaum-Wurzel, also U=12.5 ist. Da so aber keine neue kürzere Rundreise gefunden werden kann, brauchen wir den gesamten Teilbaum nicht weiter zu betrachten.

Weiter zurückgegangen im Entscheidungsbaum kommen wir zu der Verzweigung, nach der wir uns entschieden haben „nicht von 0 nach 1" zu gehen. Anstatt „von 0 nach 2" zu gehen, können wir uns entscheiden „nicht von 0 nach 2" zu gehen. Allerdings liegt die untere Schranke hierfür bei U=12, ist also größer als die aktuelle obere Schranke O=11. D.h. wie im vorigen Fall, dass wir den gesamten Teilbaum „abschneiden" können und nicht weiter betrachten müssen.

Ein weiteres Zurückgehen im Baum bringt uns wieder zur Ausgangssituation (der Wurzel des Baumes). Hier haben wir allerdings beide Entscheidungen („von 0 nach 1“ / „nicht von 0 nach 1“) schon abgehandelt, womit der Algorithmus terminiert.

Nach Beendigung des Verfahrens haben wir eine minimale Rundreise, also die Lösung für dieses TSP, gefunden: die erste gefundene minimale Rundreise war: 0 → 1 → 2 → 3, die Kosten hierfür werden durch die obere Schranke beschrieben: O=11.

Was hat uns der Einsatz von Branch-and-Bound speziell in diesem Fall gebracht? Dies verdeutlicht die folgende Skizze des binären Entscheidungsbaumes für das Problem. Jeder Knoten des Baumes ist dabei stellvertretend für die Situation nach der jeweiligen Entscheidung:



Mit dem Algorithmus des „erschöpfenden Durchsuchens“, also des Findens aller Permutationen der Knoten des Graphen, hätten wir den gesamten Entscheidungsbaum durchlaufen und uns alle Rundreisen betrachten müssen (6 Stück).
Die grünen Pfade und Knoten stellen nun die Wege dar, die wir betrachten mussten, um mit Branch-and-Bound zu einer Lösung des TSP zu gelangen, die roten Pfade und Knoten brauchten wir dabei nicht betrachten, da wir diese Teilbäume abschneiden konnten.

Allgemein ist festzuhalten, dass die Komplexität des Problems TSP durch Lösung mittels Branch-and-Bound im schlechtesten Fall nicht über die triviale Lösung des Aufzählens aller Permutationen hinauskommt (siehe auch 6.). Allerdings ist zu bedenken, dass dieser schlechteste Fall nur in seltenen Fällen auftritt und sich mittels Branch-and-Bound in den anderen Fällen viel Laufzeit einsparen lässt.

4. Algorithmus-Eigenschaften

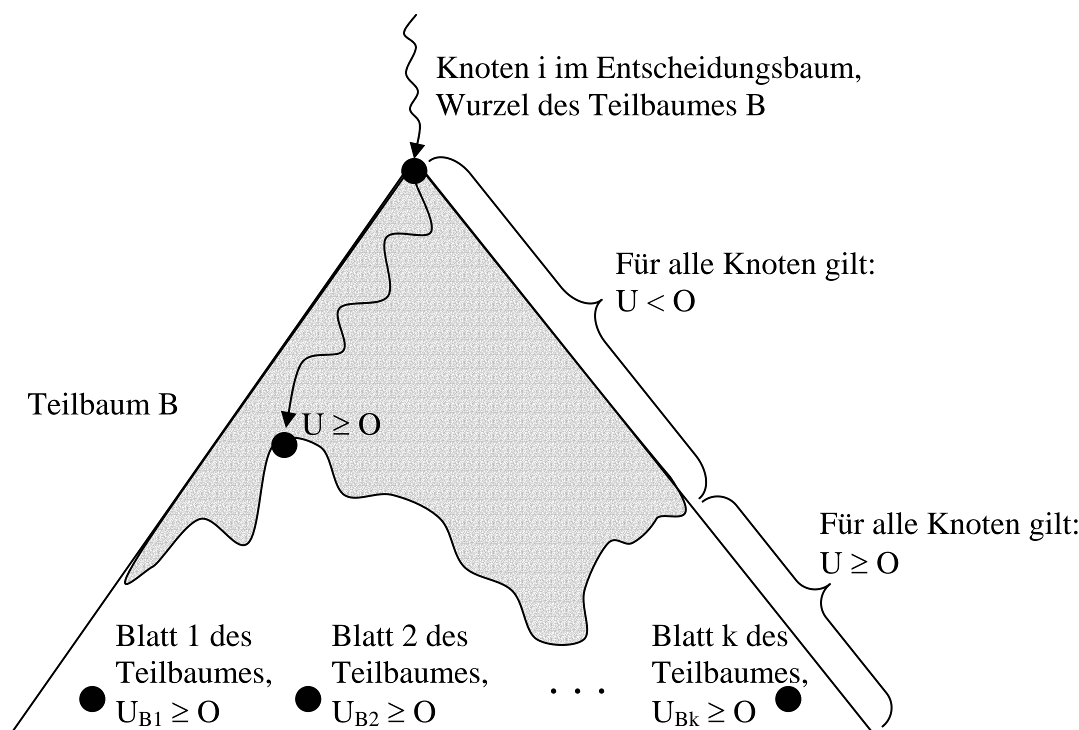
4.1 Traversierung des Entscheidungsbaumes:

Wie bereits in 2. klar geworden sein sollte, wird der Entscheidungsbaum so abgelaufen, dass in jedem Baumknoten zwischen den Fällen 1.) „$U_1 \leq U_2$“, 2.) „$U_1 > U_2$“ und 3.) „B_2 existiert nicht“ unterschieden wird. Je nachdem, welcher Fall zutrifft, wird zuerst in den linken Teilbaum B_1 bzw. in den rechten Teilbaum B_2 verzweigt und danach in den jeweils anderen Teilbaum. Sollte B_2 nicht existieren, so wird nur nach B_1 verzweigt. Sollten eine untere Schranke größer gleich der aktuellen oberen Schranke sein, so wird im Bounding-Schritt der darunter liegende Teilbaum „abgeschnitten“ und nicht weiter betrachtet, da die darin enthaltene Teillösungsmenge zu keiner kürzeren Rundreise führen kann.

Bei der Traversierung von Teilbäumen kann man wiederum 2 Fälle unterscheiden:

(a) Es wird keine kürzere Rundreise gefunden:

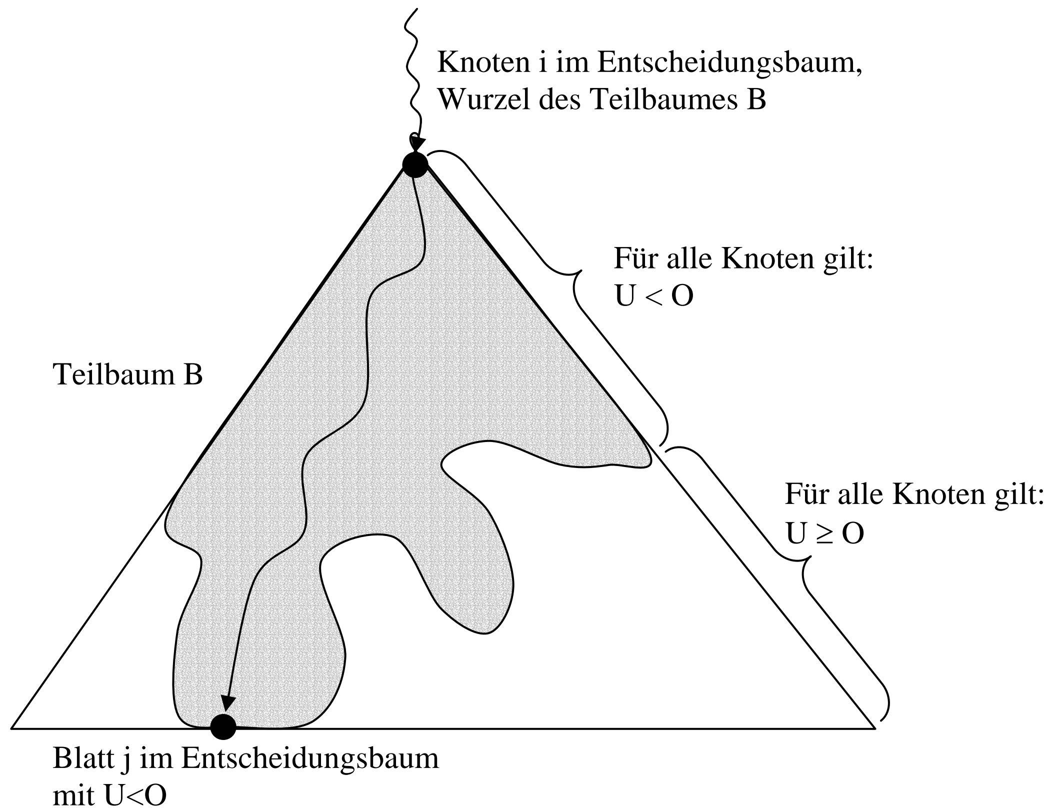
In diesem Fall führen alle Wege durch den Teilbaum in eine „Sackgasse“, d.h. darunter liegende Teilbäume werden aufgrund einer unteren Schranke $U \geq O$ abgeschnitten. Das bedeutet auch, dass alle im Teilbaum enthaltenen Blätter (also die im Teilbaum enthaltene Teillösungsmenge) eine untere Schranke $U \geq O$ besitzen und somit im Teilbaum keine kürzere Rundreise als die bisher kürzeste vorhanden ist. Zur Illustration wieder eine Grafik:



Für alle Knoten des grau unterlegten Bereichs des Teilbaumes gilt $U<O$, sodass wir mit der Hoffnung, eine neue kürzere Rundreise zu finden, weiter in die Tiefe gehen können.
Doch diese Hoffnung stellt sich als Trugschluss heraus, da für alle Blätter des Teilbaumes, die ja Rundreisen darstellen, $U \geq O$ gilt und es somit keine kürzere Rundreise im Teilbaum geben kann.

(b) Es wird eine kürzere Rundreise gefunden als die bisher kürzeste:

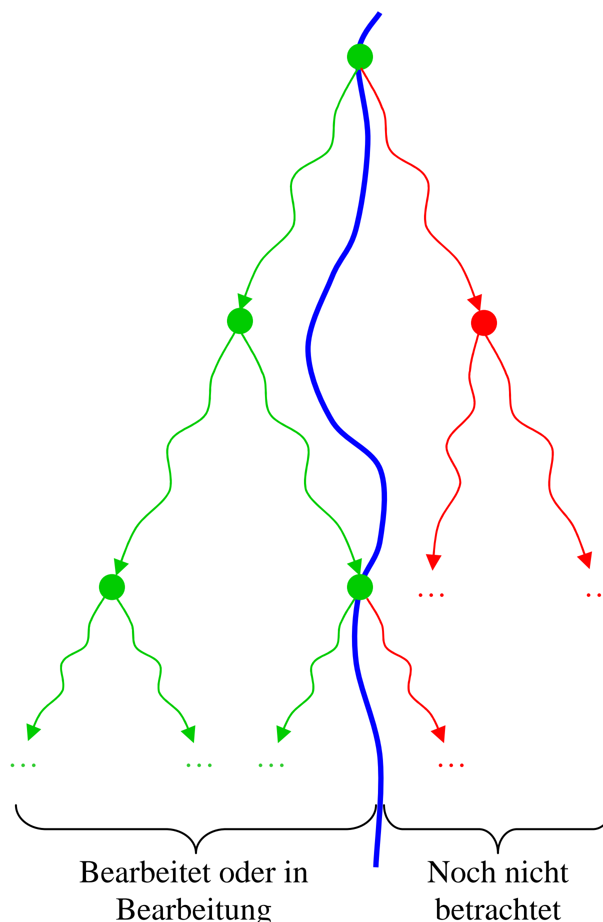
In diesem Fall gibt es mind. ein Blatt im betrachteten Teilbaum, welches eine untere Schranke U kleiner der bisherigen oberen Schranke O besitzt. Folgende Grafik veranschaulicht die Situation:



Die Konsequenzen hieraus sind eine neue kürzere Rundreise, die man sich merkt und deren Länge (untere Schranke U) die neue obere Schranke O bestimmt. Alle noch nicht betrachteten Teilbäume, die keine untere Schranke kleiner der neuen oberen Schranke O besitzen, brauchen nicht weiter betrachtet werden.

Wie bereits in 2. erwähnt wurde, werden die beiden Schritte Branch und Bound solange ausgeführt, bis der gesamte Entscheidungsbaum traversiert wurde, indem man kürzere Rundreisen gefunden hat bzw. indem Teilbäume abgeschnitten wurden, da sie eine untere Schranke $U \geq O$ besessen haben.
In jeder Situation kann man dabei festhalten, welcher Teil des Entscheidungsbaumes bereits betrachtet wurde und welcher Teilbäume noch abzuarbeiten sind. Dies lässt sich grafisch als Baum so darstellen, dass wir in Teilbäumen, wo gerade ein Sohn S abgearbeitet wird und der zweite Sohn T noch nicht betrachtet wurde, den „aktiven“ Sohn S als linken und den „inaktiven“ Sohn S als rechten Sohn anordnen. Somit erhalten wir einen Baum, in dem bereits abgearbeitete bzw. gerade bearbeitete Teilbäume „links“ und noch nicht betrachtete Teilbäume „rechts“ von einem Schnitt durch den Baum angeordnet sind.

Folgende Grafik illustriert die Situation:



4.2 <u>Korrekter Aufbau des Entscheidungsbaumes:</u>

Allgemein kann man festhalten, dass der Entscheidungsbaum korrekt aufgebaut wird, wenn sich mit diesem alle (n-1)! Permutationen (Rundreisen durch die Städte) erzeugen lassen können.

Dies ist der Fall, wenn die Distanzmatrix korrekt gesetzt wird und korrekte Bestimmungen über den weiteren Aufbau des Entscheidungsbaumes getroffen werden.

Wie in 2. bereits betrachtet wurde, geschieht der Aufbau des Entscheidungsbaumes derart, dass bei jedem Baum-Knoten mit der dort gültigen Distanzmatrix (gültig heißt: mit dem aktuellen Anfangsstück der Rundreise konform) geprüft wird, welche Städte von der aktuellen Stadt i aus erreichbar sind.
Es wurde betrachtet, dass dabei 3 Situationen unterschieden werden können, auf die ich hier noch einmal ausführlicher eingehen will:

a) <u>Das aktuelle Anfangsstück der Rundreise hat die Länge n-1:</u> in diesem Fall gehen wir von i zurück zum Knoten j=0 und komplettieren somit die Rundreise.
b) <u>Für die Menge I der von i aus erreichbaren Städte gilt |I\{0}| > 1:</u> das bedeutet, dass mehr als eine Stadt von i aus erreichbar ist. Wir wählen die zahlenmäßig kleinste Stadt j aus I aus und führen eine Fallunterscheidung durch zwischen: A) „Setze Rundreise über j fort“, B) „Setze Rundreise nicht über j fort“. Für beide Entscheidungen sind neue Distanzmatrizen zu bilden, indem bei A) die Kante (i,j) inkludiert und bei B) die Kante (i,j) ausgeschlossen wird (s.u. für Erläuterungen).

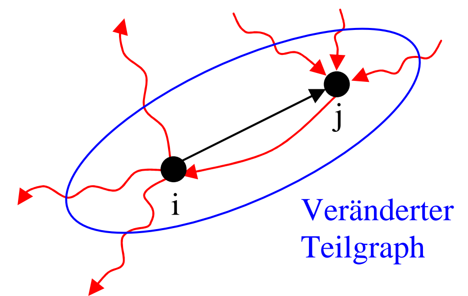
c) Für die Menge I der von i aus erreichbaren Städte gilt |I\{0}| = 1: das bedeutet, dass uns keine Wahl bleibt – wir müssen das aktuelle Anfangsstück der Rundreise über den einzigen in I vorhandenen Knoten j fortsetzen. Die Distanzmatrix für den Teilbaum der Entscheidung wird durch Inkludieren der Kante (i,j) aktualisiert (s.u. für Erläuterungen).

Aktualisierungen der Distanzmatrix:

- Inkludieren der Kante (i,j):
  Das bedeutet, dass das aktuelle Anfangsstück der Rundreise über Knoten j fortgesetzt wird. Um die Distanzmatrix korrekt zu setzen, müssen alle Kanten, die dadurch für eine Rundreise nicht mehr gangbar sind, entnommen werden.
  Dies sind:
  1.) alle in j eingehenden Kanten bis auf (i,j), da wir von i nach j gehen und dadurch nicht von einem anderen Knoten nach j gehen können,
  2.) alle aus i ausgehenden Kanten außer (i,j), da wir von i nach j gehen und dadurch nicht von i zu einem anderen Knoten gehen können und
  3.) die Kante (j,i), da der Gang von i nach j die Kante von j nach i invalidiert.
  Die Aktualisierung der Distanzmatrix gestaltet sich folgendermaßen (x bedeutet: das Element/die Teilmatrix wird nicht verändert):

$dist_A =$

	0	...	i	...	j	...	n-1
0			x		∞		x
...							
i			∞		x		∞
...							
j	x		∞		∞		x
...							
n-1							

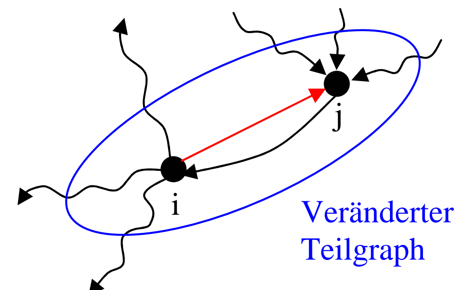


  Im rechten Teilbild spiegeln die roten Pfeile alle Kanten wider, die durch das Inkludieren von (i,j) ausgeschlossen werden.
  Beim Inkludieren wird somit sichergestellt, dass j zum Anfangsstück der Rundreise hinzugefügt wird. j kann nun von keinem anderen Knoten mehr erreicht werden, die Rundreise kann allerdings von j aus über alle Knoten ungleich i fortgesetzt werden.

- Ausschließen der Kante (i,j):
  Das bedeutet, dass das aktuelle Anfangsstück der Rundreise nicht über Knoten j fortgesetzt wird. Um die Distanzmatrix korrekt zu setzen, ist nur die Kante (i,j) zu entnehmen, in der Distanzmatrix ist also das Element $dist_{i,j}=\infty$ zu setzen:

$dist_A =$

	0	...	j	...	n-1
0	x				
...					
i			∞		
...					
n-1					



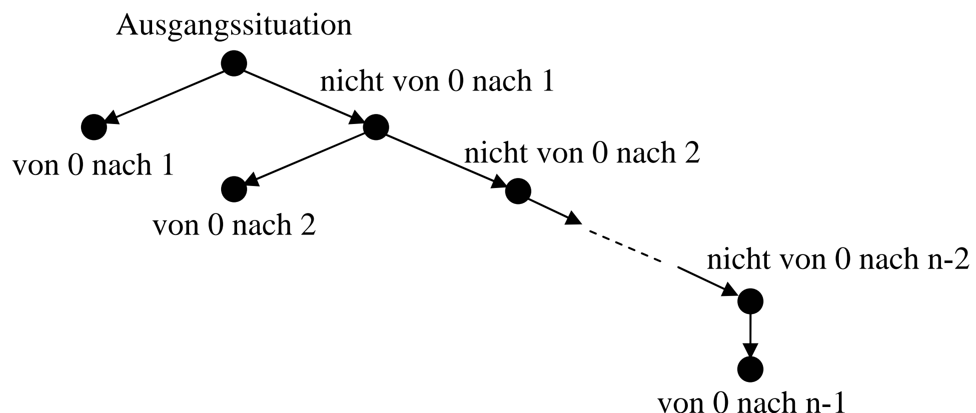
  Beim Ausschließen wird somit sichergestellt, dass die Rundreise nicht über den Knoten j fortgesetzt wird und ein anderer Knoten von i aus gesucht werden muss. j kann weiterhin von allen Knoten außer i benutzt werden, um darüber eine Rundreise fortzusetzen.

Mit diesen Aktualisierungen lässt sich nachweisen, dass eine Distanzmatrix mit einem Anfangsstück der Länge (n-1) eine Permutation der n Städte enthält. Dazu muss (n-1) mal eine Kante (i,j) inkludiert werden.

Bei diesem Schritt werden (n-1) mal Zeilen und Spalten auf ∞ gesetzt bis auf Elemente (i,j), die wir als Kanten in die Rundreise inkludiert haben. Aneinandergereiht bilden diese Elemente eine Permutation der n Städte.
Bei den (n-1) Inkludierungen einer Kante (i,j) wird jede Zeile genau einmal betrachtet. D.h. kein i zweimal. Dies ergibt sich daraus, dass bei der vorherigen Inkludierung einer Kante (k,i) alle in i eingehenden Kanten bis auf die zur Rundreise gehörige auf ∞ gesetzt wurden.
Ebenso wird eine Spalte genau einmal betrachtet, d.h. kein j zweimal. Würde eine Spalte mehrmals betrachtet werden, so müsste es mindestens 2 Elemente ≠∞ in Spalte j geben. Durch Inkludieren von (i,j) bleibt aber nur Element i in Spalte j erhalten.

Wird der Entscheidungsbaum nun so aufgebaut, dass sich alle (n-1)! Permutationen erzeugen lassen? Das lässt sich induktiv beweisen:

a) Induktionsanfang:
Mit dem Entscheidungsbaum werden alle (n-1) Anfangsstücke der Länge 1 konstruiert.
Begründung: die Konstruktion aller Anfangsstücke der Länge 1 geschieht durch Fallunterscheidung vom Startknoten 0 aus. Von diesem existieren (n-1) Möglichkeiten, zu einem Folgeknoten zu gelangen, d.h. (n-2) mal lässt sich ein linker und rechter Sohn konstruieren. Gehen wir nach rechts, so ist durch Ausschluss der Kante (0,k) mit 0<k≤n-2 der nächste von 0 aus erreichbare Knoten der Knoten (k+1), sodass die nächste Entscheidung lautet, entweder nach (i+1) oder nicht nach (i+1) zu gehen.
Durch den Gang zum jeweils linken Sohn an allen so erzeugten Entscheidungsknoten werden alle (n-1) Anfangsstücke der Länge 1 konstruiert.
Folgende Grafik illustriert den Sachverhalt:



b) Induktionsvoraussetzung:
Es sind alle Anfangsstücke der Länge i vorhanden (1 ≤ i ≤ n-2),
insgesamt (n-1)*(n-2)*…*(n-i) Stück.
c) Induktionsbehauptung:
Aus den Anfangsstücken der Länge i lassen sich alle Anfangsstücke der Länge (i+1) konstruieren, insgesamt (n-1)*(n-2)*…*(n-i-1) Stück.
d) Induktionsschritt:
Mit einem Anfangsstück der Länge i gibt es (n-i-1) Möglichkeiten, einen Nachfolgeknoten zu finden. Begründung: ausgehend vom vollständigen Problemgraphen wurden mit der Aktualisierung der Distanzmatrix durch Inkludieren einer Kante (k,l) (1 ≤ k,l ≤ n-1, k≠l) alle in l eingehenden Kanten invalidiert. Da dies bei einem Anfangsstück der Länge i i-mal geschehen kann, kommen (n-1)-i Nachfolgeknoten in Frage.
D.h. dass (n-2)-i mal von einem Anfangsstück der Länge i aus im Entscheidungsbaum ein linker und ein rechter Sohn erzeugt werden kann.

Das bedeutet, dass von jedem dieser Anfangsstücke (n-1)-i Anfangsstücke der Länge i+1 gefunden werden können, insgesamt damit (n-1)*(n-2)*…*(n-i)*(n-i-1) Stück.

Für i=n-1 ergibt sich dadurch: Es existieren (n-1)*(n-2)*…*(n-1-n+2) = (n-1)! Anfangsstücke, wobei durch die Länge (n-1) dieser jeweils eine Rundreise komplettiert wird. D.h. mit dem Entscheidungsbaum werden (n-1)! Rundreisen erzeugt → genau das war nachzuweisen.

4.3 <u>Algorithmus-Invarianten:</u>

Eine Invariante für einen Aufruf von „Branch“ besteht darin, dass die obere Schranke O stets gleich der bis dahin gefundenen minimalen Kosten einer Rundreise ist (mit der Bedingung, dass vor dem ersten Finden einer Rundreise $O = \infty$ gilt).

Dies ergibt sich aus der Traversierung des Entscheidungsbaumes. Bei dieser werden alle Knoten betrachtet, welche eine untere Schranke $U < O$ (der aktuellen oberen Schranke) haben. Darin sind auch alle Blätter des Entscheidungsbaumes eingeschlossen, die zu kürzeren Rundreisen führen als bisher gefunden. Ist also die Länge des Anfangsstücks einer Rundreise in einem Knoten des Entscheidungsbaumes (n-1) und gilt für die untere Schranke dieses Knotens $U < O$, so wurde eine neue kürzere Rundreise gefunden und $O = U$ gesetzt.

4.4 <u>Bedingungen für den Algorithmus:</u>

- Vorbedingung: Es liegt eine korrekte Distanzmatrix des Problems vor.
- Nachbedingung: Es wurde eine minimale Rundreise gefunden, O enthält ihre Kosten.

4.5 <u>Korrektheit des Algorithmus:</u>

Um die Korrektheit nachzuweisen, ist zu prüfen, ob bei erbrachter Vorbedingung nach Abarbeitung des Algorithmus schließlich die Nachbedingung gilt.

Wenn man alle (n-1)! Permutationen betrachten würde, so würde man auf jeden Fall die kürzeste Rundreise daraus auswählen können. D.h. wenn durch den Entscheidungsbaum alle (n-1)! Permutationen erzeugt werden könnten und eine vollständige Traversierung des Baumes stattfinden würde, so käme man nachweislich zu einer Lösung des TSP. Dass sich mit dem Entscheidungsbaum (n-1)! Permutationen erzeugen lassen, wurde in 4.2 bewiesen. Durch Traversierung des Baumes mit einer konstanten oberen Schranke $O=\infty$ würden alle diese Lösungen betrachtet werden, sodass der Algorithmus in diesem Fall korrekt arbeitet.

Im allgemeinen Fall muss noch bewiesen werden, dass das Abschneiden von Teilbäumen bei der in 4.1 besprochenen Traversierung korrekt ist, also dass in diesen Teilbäumen keine Rundreise enthalten sein kann, die kürzer der bisher kürzesten Rundreise sein kann. Die Länge dieser ist wie in 4.3 gezeigt zu jedem Zeitpunkt in der oberen Schranke O festgehalten. Es verhält sich nun so, dass (wie bereits in 4.1 besprochen) Teilbäume nur in dem Fall abgeschnitten werden, wo für deren untere Schranke U gilt: $U \geq O$. Dass dies korrekt ist, lässt sich schnell zeigen: denn U stellt eine untere Schranke für den gesamten Teilbaum dar und für alle Knoten i des Teilbaumes lässt sich festhalten: $U_i \geq U$ und damit auch $U_i \geq O$. Eine kürzere Rundreise muss aber die Bedingung $U < O$ erfüllen und kann damit nicht im abgeschnittenen Teilbaum enthalten sein.

Mit der Traversierung des Entscheidungsbaumes von 4.1, der Korrektheit des Aufbaus des Entscheidungsbaumes von 4.2 und den eben dargelegten Schlussfolgerungen kann man sagen, dass der Algorithmus korrekt arbeitet.

4.6 Termination des Algorithmus:

Die Termination des Algorithmus ergibt sich aus der Traversierung des Entscheidungsbaumes (s. 4.1) und aus der Korrektheit seines Aufbaus (s. 4.2). Da der Baum nur endlich viele Knoten besitzen kann (er besitzt wie in 4.2 gezeigt mit (n-1)! nur endlich viele Blätter), jeder Baumknoten höchstens einmal betrachtet wird und in jedem Durchlauf von Branch-and-Bound genau ein Knoten untersucht wird, terminiert der Algorithmus in endlich vielen Schritten.

5. Exkurs: Rechnen mit +∞ und -∞

5.1. Mathematisches Rechnen mit +∞ und -∞:

Bei der Rechnung mit natürlichen bzw. verallgemeinert mit reellen Zahlen gibt zwar keine Zahl „unendlich", allerdings kann man sich Abhilfe verschaffen, indem man ein Symbol „∞" mit Hilfe der Grenzwertrechnung als Grenzwert einer reellen Zahlenfolge wie folgt einführt:

$$\lim_{k \to \infty} a_k = \infty$$, a_k…Folge reeller Zahlen, k>0

Ebenso lässt sich ein Symbol „-∞" definieren:

$$\lim_{k \to \infty} a_k = -\infty$$

Für die reellen Zahlen seien dann „∞" sowie „-∞" so definiert, dass folgende Ungleichung gilt:

$$-\infty < x < \infty, \quad \forall x \in R$$

Damit überträgt sich sofort die Ordnungsrelation „<" von der Menge der reellen Zahlen ℝ auf die Menge ℝ∪{-∞,∞}.

Die meisten Rechenregeln für arithmetische Operationen übertragen sich von ℝ auf ℝ∪{-∞,∞}, speziell die Rechnungen für Unendlichkeiten seien nachfolgend weiter betrachtet:

Addition:

$$\lim_{x \to \infty} x + \lim_{y \to \infty} y = \infty + \infty = \infty$$

$$a + \lim_{x \to \infty} x = \lim_{x \to \infty} (a + x) = a + \infty = \infty, \ \forall a \in R$$

$$\lim_{x \to -\infty} x + \lim_{y \to -\infty} y = -\infty - \infty = -\infty$$

$$a + \lim_{x \to -\infty} x = \lim_{x \to -\infty} (a + x) = a - \infty = -\infty, \ \forall a \in R$$

→ Assoziativität, Kommutativität und Distributivität ergeben sich aus der Addition in ℝ.

Subtraktion:

$$\lim_{x\to-\infty} x - \lim_{y\to\infty} y = -\infty - \infty = -\infty$$

$$a - \lim_{x\to\infty} x = \lim_{x\to\infty}(a - x) = a - \infty = -\infty,\ \forall a \in R$$

$$\lim_{x\to\infty} x - \lim_{y\to-\infty} y = \infty + \infty = \infty$$

$$a - \lim_{x\to-\infty} x = \lim_{x\to-\infty}(a - x) = a + \infty = \infty,\ \forall a \in R$$

→ Assoziativität, Kommutativität und Distributivität ergeben sich aus der Subtraktion reeller Zahlen.

Multiplikation:

$$\lim_{x\to\infty} x \cdot \lim_{y\to\infty} y = \infty \cdot \infty = \infty$$

$$\lim_{x\to-\infty} x \cdot \lim_{y\to-\infty} y = (-\infty) \cdot (-\infty) = \infty$$

$$\lim_{x\to-\infty} x \cdot \lim_{y\to\infty} y = (-\infty) \cdot \infty = -\infty$$

$$a \cdot \lim_{x\to\infty} x = \lim_{x\to\infty}(a \cdot x) = a \cdot \infty = \infty,\ \forall a \in R_{>0}$$

$$a \cdot \lim_{x\to\infty} x = \lim_{x\to\infty}(a \cdot x) = a \cdot \infty = -\infty,\ \forall a \in R_{<0}$$

$$a \cdot \lim_{x\to-\infty} x = \lim_{x\to\infty}(a \cdot x) = a \cdot (-\infty) = -\infty,\ \forall a \in R_{>0}$$

$$a \cdot \lim_{x\to-\infty} x = \lim_{x\to\infty}(a \cdot x) = a \cdot (-\infty) = \infty,\ \forall a \in R_{<0}$$

→ Assoziativität, Kommutativität und Distributivität ergeben sich aus der Multiplikation reeller Zahlen.

Division:

$$a / \lim_{x\to\infty} x = \lim_{x\to\infty}(a / x) = a / \infty = 0,\ \forall a \in R_{\neq 0}$$

$$a / \lim_{x\to-\infty} x = \lim_{x\to-\infty}(a / x) = a / (-\infty) = 0,\ \forall a \in R_{\neq 0}$$

$$a / 0 = \lim_{x\to a} x / \lim_{y\to 0} 0 = \infty$$

→ Assoziativität ergibt sich aus der Division reeller Zahlen.

Doch nicht alle arithmetischen Operationen sind definiert. Folgende Auflistung gibt diese an und zeigt anhand der Rechnung mit Grenzwerten, warum sie undefiniert sind:

$$\lim_{x\to\infty} x + \lim_{y\to-\infty} y = \infty - \infty = n.d.$$

$$\lim_{x\to\infty} x - \lim_{y\to\infty} y = \infty - \infty = n.d.$$

$$\lim_{x\to-\infty} x - \lim_{y\to-\infty} y = (-\infty) - (-\infty) = n.d.$$

$$\lim_{x\to\infty} x \cdot \lim_{y\to 0} y = \infty \cdot 0 = n.d.$$

$$\lim_{x\to-\infty} x \cdot \lim_{y\to 0} y = (-\infty) \cdot 0 = n.d.$$

$$\lim_{x\to\infty} x / \lim_{y\to 0} y = \infty / 0 = n.d.$$

$$\lim_{x\to 0} x / \lim_{y\to\infty} y = 0 / \infty = n.d.$$

$$\lim_{x\to\infty} x / \lim_{y\to\infty} y = \infty / \infty = n.d.$$

$$\lim_{x\to-\infty} x / \lim_{y\to\infty} y = (-\infty) / \infty = n.d.$$

$$\lim_{x\to-\infty} x / \lim_{y\to-\infty} y = \infty / \infty = n.d.$$

$$\lim_{x\to 0} x / \lim_{y\to 0} y = 0 / 0 = n.d.$$

→ Alle diese Fälle laufen auf einen unbestimmten Grenzwert hinaus, über den man ad hoc keine Aussage treffen kann.

5.2. Rechnen mit Unendlichkeitswerten in Java:

Für das Rechnen mit +/- ∞ im Reellen stellt Java in den beiden Klassen „Double“ und „Float“ jeweils die Attribute POSITIVE_INFINITY und NEGATIVE_INFINITY bereit für die Repräsentation von +∞ bzw. -∞. Alle in 5.1 aufgeführten mathematischen Regeln des Rechnens mit +/- ∞ übertragen sich auf diese beiden Attribute.
Speziell gilt für Vergleiche mit einer beliebigen Zahl „Double a“ (a ≠ POSITIVE_INFINITY und a ≠ NEGATIVE_INFINITY):

NEGATIVE_INFINITY < a < POSITIVE_INFINITY

Zusätzlich gelten allerdings entgegen der mathematischen Grenzwert-Rechnung folgende Operationen als definiert:

0.0/ POSITIVE_INFINITY = 0.0
0.0/ NEGATIVE_INFINITY = -0.0
POSITIVE_INFINITY/0.0 = POSITIVE_INFINITY
NEGATIVE_INFINITY/0.0 = NEGATIVE_INFINITY

Dies ist der Fall, da sich Java nach dem IEEE 754-Standard für Gleitpunktzahlen richtet und diese Operationen mit Infinity dort so definiert sind (siehe IEEE 754, Kapitel 7.1).

Für nicht definierte Ergebnisse bietet Java ebenfalls in den Klassen „Double“ und „Float“ das Attribute NaN (Not a Number) an. Dies ist das Ergebnis bei undefinierten Rechnungen (wieder dem IEEE 754-Standard folgend):

POSITIVE_INFINITY-POSITIVE_INFINITY = NaN
NEGATIVE_INFINITY-NEGATIVE_INFINITY = NaN
POSITIVE_INFINITY+NEGATIVE_INFINITY = NaN
POSITIVE_INFINITY*0.0 = NaN
NEGATIVE_INFINITY*0.0 = NaN
POSITIVE_INFINITY/ POSITIVE_INFINITY = NaN
POSITIVE_INFINITY/NEGATIVE_INFINITY = NaN
NEGATIVE_INFINITY/ POSITIVE_INFINITY = NaN
0.0/0.0 = NaN

Für die vorliegende Lösung des TSP mit Branch-and-Bound genügt allerdings bereits der Vergleich mit POSITIVE_INFINITY, weshalb ich diese kurze Exkursion hier beenden möchte.

6. Praktische Umsetzung

Allgemein ist zu bemerken, dass die Distanzmatrix als „double“ abgespeichert wird, also reelle Zahlen enthalten soll. Dies ist nötig, um auf das Attribut Double.POSITIVE_INFINITY zurückgreifen zu können.

Folgende Java-Klasse soll den oben beschriebenen Algorithmus korrekt umsetzen:

```
1 public class TSP {
2   public double m_upperBound;
3   public double[][] m_shortestKnownTour;
4
5   double lowerBound(double[][] distMatrix){
6           double lBound = 0.0;
7           for(int i=0; i<distMatrix.length; i++){
8                   double rowMin = Double.POSITIVE_INFINITY;
9                   for(int j=0; j<distMatrix[i].length; j++){
10                          rowMin = Math.min(rowMin, distMatrix[i][j]);
11                  }
12                  lBound += rowMin;
13          }
14          for(int j=0; j<distMatrix[0].length; j++){
15                  double colMin = Double.POSITIVE_INFINITY;
16                  for(int i=0; i<distMatrix.length; i++){
17                          colMin = Math.min(colMin, distMatrix[i][j]);
18                  }
19                  lBound += colMin;
20          }
21          return lBound/2.0;
22  }
23
24  void includeEdge(int from, int to, double[][] distMatrix){
25          for(int j=0; j<distMatrix[from].length; j++){
26                  if(j != to){
27                          distMatrix[from][j] = Double.POSITIVE_INFINITY;
28                  }
29          }
30          for(int i=0; i<distMatrix.length; i++){
31                  if(i != from){
32                          distMatrix[i][to] = Double.POSITIVE_INFINITY;
33                  }
34          }
35          if(distMatrix.length != 2){
36                  distMatrix[to][from] = Double.POSITIVE_INFINITY;
37          }
38  }
39
40  void excludeEdge(int from, int to, double[][] distMatrix){
41          distMatrix[from][to] = Double.POSITIVE_INFINITY;
42  }
```

```
43
44
45  int getNumberOfReachableNodes(int from, double[][] distMatrix){
46          int nodes = 0;
47          for(int j=0; j<distMatrix.length; j++){
48                  if(distMatrix[from][j] < Double.POSITIVE_INFINITY){
49                          nodes++;
50                  }
51          }
52          return nodes;
53  }
54
55  int getFirstReachableNode(int from, double[][] distMatrix){
56          int node = -1;
57          for(int j=0; j<distMatrix[from].length && node==-1; j++){
58                  if(distMatrix[from][j] < Double.POSITIVE_INFINITY){
59                          node = j;
60                  }
61          }
62          return node;
63  }
64
65  void branchAndBound(int actTourLength, int fromNode, double[][] distMatrix){
66          double u1 = Double.POSITIVE_INFINITY;
67          double u2 = Double.POSITIVE_INFINITY;
68          double[][] distMatrix1 = null;
69          double[][] distMatrix2 = null;
70
71          if(actTourLength == distMatrix.length-1){
72                  double lBound = lowerBound(distMatrix);
73
74                  if(lBound < m_upperBound){
75                          m_upperBound = lBound;
76                          m_shortestKnownTour = distMatrix;
77                  }
78                  return;
79          }
80
81          distMatrix[fromNode][0] = Double.POSITIVE_INFINITY;
82
83          int reachableNodes = getNumberOfReachableNodes(fromNode, distMatrix);
84          int reachableFirst = getFirstReachableNode(fromNode, distMatrix);
85
86          if(reachableNodes > 1){
87                  distMatrix2 = cloneDistMatrix(distMatrix);
88                  excludeEdge(fromNode, reachableFirst, distMatrix2);
89                  u2 = lowerBound(distMatrix2);
90          }
91          distMatrix1 = distMatrix;
92          includeEdge(fromNode, reachableFirst, distMatrix1);
93          u1 = lowerBound(distMatrix1);
94
95          if(u1 <= u2){
96                  if(u1 < m_upperBound){
97                          branchAndBound(actTourLength+1, reachableFirst, distMatrix1);
98                          if(u2 < m_upperBound){
99                                  branchAndBound(actTourLength, fromNode, distMatrix2);
100                         }
101                 }
102         }else{
103                 if(u2 < m_upperBound){
104                         branchAndBound(actTourLength, fromNode, distMatrix2);
105                         if(u1 < m_upperBound){
106                                 branchAndBound(actTourLength+1, reachableFirst, distMatrix1);
107                         }
108                 }
109         }
110 }
111
112 public void solveTSP(double[][] distMatrix){
113         if(distMatrix.length<2){
114                 return;
115         }
116
117         m_upperBound = Double.POSITIVE_INFINITY;
118         m_shortestKnownTour = null;
119
120         branchAndBound(0, 0, distMatrix);
121 }
122
```

```
123 double[][] cloneDistMatrix(double[][] distMatrix){
124         double[][] clone = distMatrix.clone();
125         for(int i=0; i<distMatrix.length; i++){
126                 clone[i] = distMatrix[i].clone();
127         }
128         return clone;
129 }
130}
```

Nachfolgend möchte ich eine ausführliche Betrachtung der einzelnen Attribute und Methoden durchführen:

(a) `public double m_upperBound` (Zeile 2):
Dies stellt die obere Schranke O dar, welche zu jedem Zeitpunkt die Länge der bisher kürzesten Rundreise enthält.

(b) `public double[][] m_shortestKnownTour` (Zeile 3):
Diese Matrix speichert die Distanzmatrix der zum jeweiligen Zeitpunkt kürzesten gefundenen Rundreise ab. In dieser Matrix sind in jeder Spalte und in jeder Zeile genau 1 Element besetzt, der Rest ist Double.POSITIVE_INFINITY. Somit kann die Rundreise aus dieser Distanzmatrix wiederhergestellt werden.

Begründung: Bei jedem Hinzufügen eines Knotens zu einem Anfangsstück der Rundreise werden nicht mehr gangbare Kanten aus dem Problemgraphen entfernt. Bei einem Anfangsstück der Länge (n-1) ist vom letzten Knoten des Stücks i aus nur noch die Kante (i,0) gangbar, da alle anderen Kanten durch die in 4.2 beschriebene Aktualisierung der Distanzmatrix invalidiert wurden.

(c) `double[][] cloneDistMatrix(double[][] distMatrix)` (Zeile 123):
- Zweck: diese Methode erstellt eine elementweise Kopie der übergebenen Distanzmatrix, da Java keinen Mechanismus bereitstellt, der eine 2D-Matrix kopiert.
- Vorbedingung: distMatrix!=null.
- Nachbedingung: Kopie der übergebenen Matrix wurde zurückgegeben.
- Zunächst wird in Zeile 124 ein Klon der Matrixzeilen erstellt. Dieser enthält aber nur die Adressen der einzelnen Zeilen der Original-Matrix. Daher werden in den Zeilen 125-127 noch die jeweiligen Zeilen (also jeweils eine 1D-Matrix) geklont, also von der Ausgangsmatrix unabhängiger Speicher erstellt.

(d) `double lowerBound(double[][] distMatrix)` (Zeile 5):
- Zweck: diese Methode berechnet die untere Schranke der Distanzmatrix und gibt diese zurück.
- Vorbedingung: distMatrix ist eine gültige Distanzmatrix, enthält also pro Zeile und Spalte mindestens ein Element !=Double.POSITIVE_INFINITY. Es gilt distMatrix.length>1.
- Nachbedingung: Die untere Schranke für die Distanzmatrix wurde berechnet und zurückgegeben. Der Rückgabewert ist:

$$U = \frac{1}{2} * \sum_{i=0}^{n-1} \left(\min_{j \in I \setminus \{i\}} dist_A(i,j) + \min_{j \in I \setminus \{i\}} dist_A(j,i) \right)$$

- In den Zeilen 7-13 wird zunächst

$$\sum_{i=0}^{n-1} \left(\min_{j \in I \setminus \{i\}} dist_A(i,j) \right)$$

berechnet und in „lBound“ gespeichert. Dies beschreibt, welcher Weg mindestens zurückgelegt werden muss, um in jede Stadt hineinzukommen.
Effektiv wird das durch Aufsummieren der Minima aller Zeilen der Distanzmatrix realisiert. Dazu wird von jeder Zeile das Minimum berechnet und in „rowMin“ gespeichert. Dieses Minimum wird dann mit „lBound“ summiert. Da in Zeile 6 „lBound=0“ gesetzt wird, enthält es nach Zeile 13 die gewünschte Summe aller Zeilenminima.

Im zweiten Schritt in den Zeilen 14-20 wird

$$\sum_{i=0}^{n-1}\left(\min_{j\in I\setminus\{i\}} dist_A(j,i)\right)$$

berechnet und zu „lBound“ addiert. Dies beschreibt, welcher Weg mindestens zurückgelegt werden muss, um aus jeder Stadt hinauszukommen.
Effektiv wird das durch Aufsummieren der Minima aller Spalten der Distanzmatrix realisiert. Dazu wird von jeder Spalte das Minimum berechnet und in „colMin“ gespeichert. Dieses Minimum wird dann zu „lBound“ addiert.
Mit der zuerst erzeugten Summe der Zeilenminima enthält „lBound“ nach Zeile 20 die Summe über alle Zeilen- und Spaltenminima, also:

$$\sum_{i=0}^{n-1}\left(\min_{j\in I\setminus\{i\}} dist_A(i,j)+\min_{j\in I\setminus\{i\}} dist_A(j,i)\right)$$

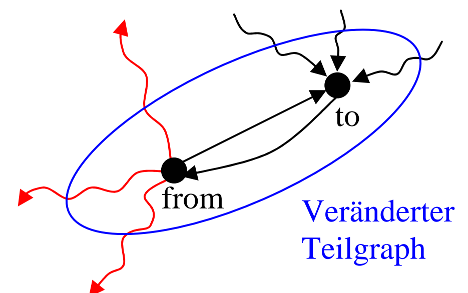
Als letztes wird diese Summe in Zeile 21 halbiert, um die Nachbedingung zu erfüllen.

(e) `void includeEdge(int from, int to, double[][] distMatrix)` (Zeile 24):

- Zweck: durch diese Methode wird die Kante from→to zum aktuellen Anfangsstück der Rundreise in der Distanzmatrix hinzugefügt.
- Vorbedingung: „from“ und „to“ sind 2 gültige Knoten des Graphen mit from!=to. „from“ muss der letzte Knoten des aktuellen Anfangsstücks der Rundreise sein, „to“ darf noch nicht Teil der Rundreise sein und muss von „from“ erreicht werden können. distMatrix muss eine gültige Distanzmatrix mit wenigstens einem Element in jeder Zeile und Spalte != Double.POSITIVE_INFINITY sein. Es gilt distMatrix.length>1.
- Nachbedingung: das in 4.2 beschriebene Inkludieren der Kante (from,to) in die Distanzmatrix muss realisiert worden sein, distMatrix muss diese Änderungen enthalten.
- Im ersten Schritt werden in den Zeilen 25-29 alle aus „from“ ausgehenden Kanten bis auf die Kante (from,to) invalidiert. Dazu werden alle Elemente der Matrixzeile „from“ mit Ausnahme des Elements „to“ auf Double.POSITIVE_INFINITY gesetzt. Dieser erste Schritt führt zu folgender Situation (x bedeutet: keine Änderung):

distMatrix =

	0	…	from	…	to	…	n-1
0				x			
…							
from		∞			x	∞	
…							
to				x			
…							
n-1							

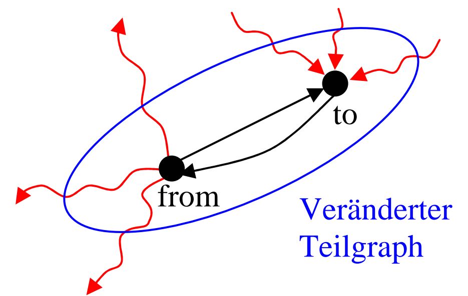


Im zweiten Schritt in den Zeilen 30-34 werden alle in „to“ eingehenden Kanten bis auf die Kante (from,to) invalidiert, indem alle Elemente der Matrixspalte „to“ mit Ausnahme des

Elements „from“ auf Double.POSITIVE_INFINITY gesetzt werden. Dieser zweite Schritt führt zu folgender Situation der Distanzmatrix:

distMatrix =

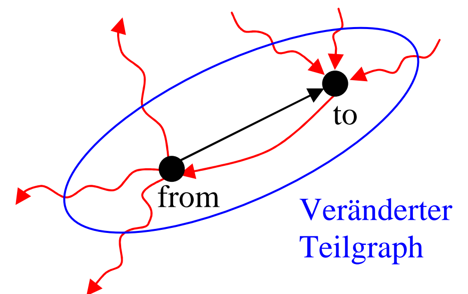
	0	...	from	...	to	...	n-1
0		x			∞		x
...							
from		∞			x		∞
...							
to		x			∞		x
...							
n-1							



Im dritten Schritt in den Zeilen 35-37 wird die Kante (to,from) invalidiert, indem dieses Matrixelement auf Double.POSITIVE_INFINITY gesetzt wird. Dies geschieht nur im Fall distMatrix.length>2. Im Fall distMatrix.length==2 darf die Kante (to,from) nicht aus dem Graphen entnommen werden, da sie die einzige Möglichkeit darstellt, vom Knoten from zurück zum Knoten to zu gelangen und somit die Rundreise zu beenden.
Die Endsituation nach Abarbeitung der Methode im Fall distMatrix.length>2 sieht dann so aus:

distMatrix =

	0	...	from	...	to	...	n-1
0		x			∞		x
...							
from		∞			x		∞
...							
to	x		∞		∞		x
...							
n-1							



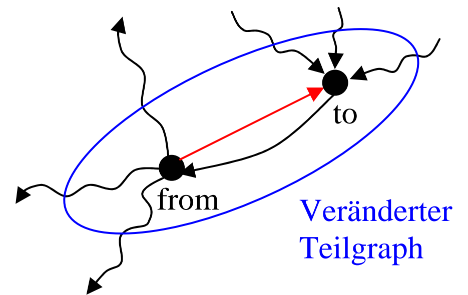
Dies entspricht der Situation nach dem Inkludieren der Kante (i,j) in 4.2 mit i=from und j=to, womit die Nachbedingung erfüllt ist.

(f) `void excludeEdge(int from, int to, double[][] distMatrix)` (Zeile 40):

- Zweck: durch diese Methode wird die Distanzmatrix so aktualisiert, dass das aktuelle Rundreisen-Anfangsstück nicht über die Kante from→to fortgesetzt wird.
- Vorbedingung: „from“ und „to“ sind 2 gültige Knoten des Graphen mit from!=to. „from“ muss der letzte Knoten des aktuellen Anfangsstücks der Rundreise sein, „to“ darf noch nicht Teil der Rundreise sein und muss von „from“ erreicht werden können. distMatrix muss eine gültige Distanzmatrix mit wenigstens einem Element in jeder Zeile und Spalte != Double.POSITIVE_INFINITY sein. Es gilt distMatrix.length>1.
- Nachbedingung: das in 4.2 beschriebene Ausschließen der Kante (from,to) muss realisiert worden sein, distMatrix muss diese Änderungen enthalten.
- Mit der Entscheidung, nicht von „from“ nach „to“ zu gehen, wird nur die Kante (from,to) invalidiert. Dies geschieht durch Setzen dieses Matrixelementes auf Double.POSITIVE_INFINITY in Zeile 40. Dadurch ergibt sich folgende Situation für die Distanzmatrix (x bedeutet keine Änderung):

distMatrix =

	0	...	to	...	n-1
0					
...	x				
from			∞		
...					
n-1					



Dies entspricht der Situation nach dem Ausschließen der Kante (i,j) in 4.2 mit i=from und j=to, womit die Nachbedingung erfüllt ist.

(g) `int getNumberOfReachableNodes(int from, double[][] distMatrix)` (Zeile 45):

- Zweck: Es soll die Anzahl aller vom Knoten „from" aus erreichbaren Knoten ermittelt und zurückgegeben werden.
- Vorbedingung: „from" ist ein gültiger Knoten des Graphen, distMatrix ist eine gültige Distanzmatrix, in der in jeder Zeile und Spalte wenigstens ein Element !=Double.POSITIVE_INFINITY vorhanden sein muss. distMatrix.length>1.
- Nachbedingung: Die Anzahl der von „from" aus erreichbaren Knoten wurde zurückgegeben.
- In Zeile 46 wird der Zähler nodes=0 gesetzt. Er soll die zurückzugebende Knoten-Anzahl enthalten. In den Zeilen 47-51 wird die Matrixzeile „from" der Distanzmatrix elementweise abgearbeitet. Ist das jeweilige Element !=Double.POSITIVE_INFINITY, so ist dieses von „from" aus erreichbar und nodes wird in dem Fall inkrementiert. Wurden alle Elemente der Zeile betrachtet, so enthält nodes dadurch am Ende die Anzahl der von „from" aus erreichbaren Knoten. Diese wird in Zeile 52 zurückgegeben.

(h) `int getFirstReachableNode(int from, double[][] distMatrix)` (Zeile 55):

- Zweck: Es soll der erste vom Knoten „from" aus erreichbare Knoten ermittelt und zurückgegeben werden.
- Vorbedingung: „from" ist ein gültiger Knoten des Graphen, distMatrix ist eine gültige Distanzmatrix, in der in jeder Zeile und Spalte wenigstens ein Element !=Double.POSITIVE_INFINITY vorhanden sein muss. distMatrix.length>1.
- Nachbedingung: Der erste von „from" aus erreichbare Knoten wurde zurückgegeben.
- In Zeile 56 wird node=-1 gesetzt. Diese Variable soll den ersten Knoten aufnehmen, der von „from" aus erreichbar ist. Nach Abarbeitung der Methode kann node==-1 nicht gelten, da in der Vorbedingung verlangt ist, dass mindestens ein Knoten von „from" aus erreichbar sein muss.
  In den Zeilen 57-61 wird dieser erste von „from" aus erreichbare Knoten ermittelt. Dazu wird die Matrixzeile „from" elementweise durchlaufen. Sollte das betrachtete Element !=Double.POSITIVE_INFINITY sein, so wird „node" auf dieses Element gesetzt und die Schleife wird abgebrochen, da node!=-1. In Zeile 62 wird dieser erste von „from" aus erreichbare Knoten zurückgegeben.

(i) `void branchAndBound(int actTourLength, int fromNode, double[][] distMatrix)` (Zeile 65):

- Zweck: Ist der Aufruf der Methode nicht-rekursiv, so soll durch diese Methode das mit der Distanzmatrix übergebene TSP gelöst werden. Anderenfalls ist ein rekursiver Aufruf erfolgt und der Entscheidungsbaum ist vom aktuellen Knoten aus gemäß des in 2. und 4. spezifizierten Algorithmus weiter aufzubauen bzw. abzuarbeiten.
- Vorbedingungen:
  I) Für nicht-rekursive Aufrufe: distMatrix ist eine gültige Distanzmatrix des zu lösenden TSP.

Für rekursive Aufrufe: distMatrix ist eine gültige Distanzmatrix, welche mit den bisher getroffenen Entscheidungen konform ist (enthält nur noch die in der Situation gangbaren Kanten) und ein Anfangsstück einer Rundreise enthält, welches in der Länge mit actTourLength übereinstimmt.

II) actTourLength muss die Länge des Anfangsstücks einer Rundreise bis zum derzeitigen Knoten im Entscheidungsbaum angeben. Es gilt: 0 <= actTourLength < distMatrix.length. Beim ersten nicht-rekursiven Aufruf der Methode muss gelten: actTourLength==0.

III) fromNode gibt den Knoten an, welcher der letzte Knoten des aktuellen Anfangsstücks der Rundreise ist und von dem aus ein Nachfolgeknoten gefunden werden soll.

IV) Die untere Schranke der durch distMatrix gegebenen Situation ist kleiner als die obere Schranke m_upperBound.

- Nachbedingungen:
  I) Für nicht-rekursive Aufrufe: m_upperBound enthält die Länge einer minimalen Rundreise, m_shortestKnownTour gibt die Distanzmatrix dieser Rundreise an. In dieser ist in jeder Spalte und Zeile jeweils genau ein Element !=Double.POSITIVE_INFINITY, sodass man die Distanzmatrix als Permutation der n Städte und somit als Rundreise ansehen kann.
  Für rekursive Aufrufe: wurde eine Rundreise gefunden, deren Kosten kleiner der bisher kürzesten Rundreise sind, so enthält m_shortestKnownTour diese kürzere Rundreise und m_upperBound ihre Länge.
- Rekursions-Invarianten:
  I) Die obere Schranke m_upperBound beinhaltet die Kosten der bis dato günstigsten Rundreise oder Double.POSITIVE_INFINITY, falls noch keine Rundreise gefunden wurde.
  II) m_shortestKnownTour beinhaltet die bis dato günstigste Rundreise oder null, falls noch keine Rundreise gefunden wurde.

Jeder Aufruf der Methode stellt eine spezifische Situation (einen konkreten Knoten) im Entscheidungsbaum dar. Wird dieser wie in 4.2 beschrieben korrekt aufgebaut und wie in 4.1 traversiert, so kann man sicher sein, nach Rückkehr aller rekursiven Aufrufe ein korrektes Ergebnis zu erhalten.

Für die Korrektheit des Aufbaus des Entscheidungsbaumes muss wie in 4.2 besprochen in der Methode zwischen drei Fällen unterschieden werden:
a) Das aktuelle Anfangsstück der Rundreise hat die Länge n-1.
b) Für die Menge I der von i aus erreichbaren Städte gilt $|I\setminus\{0\}| > 1$.
c) Für die Menge I der von i aus erreichbaren Städte gilt $|I\setminus\{0\}| = 1$.

Fall a) wird in den Zeilen 71-79 abgehandelt. Falls actTourLength == distMatrix.length-1 (Zeile 71), so wird zunächst die untere Schranke der aktuellen Situation berechnet (Zeile 72). Unter der Voraussetzung einer gültigen Distanzmatrix (Vorbedingung) ist die Berechnung dieser unteren Schranke korrekt. Sie entspricht in dieser Situation den Kosten der aktuellen Rundreise, da ein Anfangsstück mit der Länge n-1 eine Rundreise komplettiert.
In Zeile 74 wird geprüft, ob die berechnete untere Schranke kleiner der oberen Schranke m_upperBound ist. Ist dies der Fall, so gibt die aktuelle Distanzmatrix eine neue kürzere Rundreise an und m_upperBound sowie m_shortestKnownTour werden auf diese gesetzt.
Anschließend wird der Methodenaufruf bei Fall a) verlassen.
In diesem Fall bleiben die beiden Rekursions-Invarianten erhalten dadurch, dass eine neue günstigere Rundreise gefunden wurde und m_upperBound sowie m_shortestKnownTour auf diese gesetzt wurden.

Fall b) wird in den Zeilen 81 bis 109 betrachtet. Da die Länge des Anfangsstück einer Rundreise kleiner als n-1 ist, wird in Zeile 81 die Kante (fromNode,0) auf Double.POSITIVE_INFINITY gesetzt, um diese als nicht gangbar darzustellen und den Fall I\{0} abzubilden. Die Menge I\{0} der von fromNode aus erreichbaren Städte außer 0 ist gleich der Menge der Elemente, die nun in der Matrixzeile fromNode einen Wert !=Double.POSITIVE_INFINITY haben. Die Anzahl dieser wird in Zeile 83 berechnet, der erste von fromNode aus erreichbare Knoten in Zeile 84. Die Aufrufe sind gültig, da fromNode ein gültiger Knoten sein soll und distMatrix eine gültige Distanzmatrix (s. Vorbedingungen).
In Zeile 86 wird geprüft, ob |I\{0}| > 1 ist. Ist dies der Fall, so können wir zum linken und zum rechten Sohn verzweigen. Wir können also sagen: „gehe von fromNode zu rechableFirst“ oder „gehe nicht von fromNode zu reachableFirst“. Daher werden in den Zeilen 86-90 zunächst die Daten für den rechten Sohn erzeugt. Das ist die Distanzmatrix, von welcher eine Kopie erstellt wird (Zeile 87) und aus der die Kante fromNode→reachableFirst ausgeschlossen wird (Zeile 88). Im Anschluss daran wird die untere Schranke dieser transformierten Distanzmatrix bestimmt (Zeile 89). Die Methodenaufrufe sind alle korrekt, da ihre Vorbedingungen erfüllt sind.
In den Zeilen 91-93 werden die Daten für den linken Sohn erzeugt. Die Distanzmatrix wird transformiert, indem die Kante fromNode→reachableFirst inkludiert wird (Zeile 92). Im Anschluss daran wird die untere Schranke für diese Distanzmatrix berechnet.

In den Zeilen 95-109 erfolgt die weitergehende Traversierung des Entscheidungsbaumes vom aktuellen Knoten aus. Für eine korrekte Traversierung muss laut 4.1 zwischen den Fällen 1.) „$U_1 \leq U_2$“, 2.) „$U_1 > U_2$“ und 3.) „B_2 existiert nicht“ unterschieden werden.
1.) wird in den Zeilen 95-101 betrachtet. Zunächst wird in Zeile 96 geprüft, ob die linke untere Schranke u1 < m_upperBound ist. Ist dies nicht der Fall, so muss der darunter liegende Teilbaum nicht weiter betrachtet werden. Ebenso entfällt die Betrachtung des rechten Teilbaums, da gilt: u2 ≥ u1 ≥ m_upperBound. Sollte hingegen u1 < m_upperBound sein, so wird der Branch-and-Bound-Algorithmus für den linken Teilbaum rekursiv aufgerufen (Zeile 97) mit „branchAndBound(actTourLength+1, reachableFirst, distMatrix1)“. D.h. reachableFirst bildet das neue Ende des Anfangsstücks einer Rundreise, von ihm aus werden jetzt neue Entscheidungen getroffen. Nach Abarbeitung des linken Teilbaumes wird der Algorithmus rekursiv für den rechten Teilbaum aufgerufen, falls u2 < m_upperBound gilt. Ansonsten kann der rechte Teilbaum abgeschnitten werden. Der Aufruf erfolgt in Zeile 99 mit „branchAndBound(actTourLength, fromNode, distMatrix2)“, d.h. dass reachableFirst nicht zum aktuellen Anfangsstück der Rundreise hinzugefügt wurde.
Zur Korrektheit der rekursiven Aufrufe: Vorbedingung I) wird erfüllt, da distMatrix1 und distMatrix2 von includeEdge() bzw. excludeEdge() gelieferte Matrizen sind, die nach (g) und (f) korrekt gesetzt wurden. Vorbedingung II) ist erfüllt, da die Länge des Anfangsstücks nur beim Weitergehen über den linken Sohn um 1 erhöht. Vorbedingung III) ist erfüllt, da beim linken Sohn reachableFirst das neue Ende des Rundreisen-Anfangsstücks ist und beim rechten Sohn fromNode beibehalten wird. Vorbedingung IV) ist auch erfüllt durch die Tests in Zeile 96 und in Zeile 98. Da alle Vorbedingungen für branchAndBound() erfüllt sind, kann ein rekursiver Aufruf durchgeführt werden.
2.) wird in den Zeilen 102-109 betrachtet und stimmt mit Fall 1 überein, nur dass zunächst zum rechten und dann zum linken Teilbaum verzweigt wird. Rekursive Aufrufe sind dieselben und sind korrekt nach den gleichen Kriterien wie bei 1.).

3.) wird mit Fall c) des Aufbaus des Entscheidungsbaumes abgedeckt: ist |I\{0}| = 1, so werden die Zeilen 87-89 nicht ausgeführt, da der Test in Zeile 86 fehlschlägt. Das bedeutet, dass u2==Double.POSITIVE_INFINITY gilt (wie in Zeile 67 festgelegt). Die Folge daraus ist, dass die Zeilen 96-101 ausgeführt werden, dadurch dass u1 <= u2 jetzt garantiert gilt. Es wird wie bei 1.) in den linken Teilbaum verzweigt. Der Test u2 < m_upperBound in Zeile 98

schlägt fehl und der rekursive Aufruf in Zeile 99 wird nicht durchgeführt. Dies ist das in 4.2 verlangte Verhalten: der rechte Teilbaum B_2 existiert nicht, da wir mit $|I\backslash\{0\}| = 1$ nicht dahin verzweigen können. Daher wird er nicht betrachtet.

Da alle 3 Fälle a)-c) durch den Algorithmus korrekt abgearbeitet werden, wird der Entscheidungsbaum nach 4.2 korrekt aufgebaut und nach 4.1 korrekt traversiert.

Mit den Fällen b) und c) bleiben die Rekursions-Invarianten erhalten. Da die Länge des jeweiligen Rundreisen-Anfangsstücks < n-1 ist, kann in diesem Schritt keine Rundreise beendet werden und man muss tiefer in den Entscheidungsbaum verzweigen. D.h. es kann keine kürzere Rundreise gefunden werden, daher werden m_upperBound und m_shortestKnownTour nicht umgesetzt.

Fall a) erfüllt die Nachbedingung I) sofort: wie gezeigt werden bei einer neuen kürzeren Rundreise die Attribute m_upperBound und m_shortestKnownTour umgesetzt, sodass N. I) erfüllt ist.
Fall b) und c) erfüllen die Nachbedingung, da rekursiv weiter in Teilbäume des Entscheidungsbaumes verzweigt wird, bis Fall a) eintritt, womit die Nachbedingung erfüllt ist oder bis die Teilbäume untere Schranken u > m_upperBound haben und somit nicht weiter betrachtet werden müssen. In diesem Fall kann über den aktuellen Teilbaum keine kürzere Rundreise gefunden werden, womit die Nachbedingung erfüllt ist.

Ist der aktuelle Methodenaufruf nicht rekursiv erfolgt, so enthalten m_upperBound und m_shortestKnownTour die Daten einer kürzesten Rundreise durch den Graphen dadurch, dass der Entscheidungsbaum mit dem Algorithmus korrekt traversiert wurde.

(j) `public void solveTSP(double[][] distMatrix)` (Zeile 112):

- Zweck: Aufruf-Methode des Branch-and-Bound-Algorithmus zur Lösung des mit der distMatrix übergebenen TSP.
- Vorbedingung: distMatrix ist eine gültige Distanzmatrix, welche einen vollständigen gerichteten Graphen enthält. D.h. für alle Elemente (i,j) der Matrix mit $0 \leq i,j \leq n-1$, $i \neq j$ gilt: $0 \leq$ distMatrix[i][j] $<$ Double.POSITIVE_INFINITY. Weiterhin gilt für i=j: distMatrix[i][j]=Double.POSITIVE_INFINITY.
- Nachbedingung: Nach Abarbeitung des Branch-and-Bound-Algorithmus ist die Distanzmatrix einer minimalen Rundreise in m_shortestKnownTour und die Länge dieser in m_upperBound gespeichert. Die Distanzmatrix enthält genau ein Element pro Zeile und pro Spalte, sodass sie als Permutation der n Städte angesehen werden kann und somit als Rundreise.
- In den Zeilen 113-115 wird ein Schutztest durchgeführt, der prüft ob die Distanzmatrix mindestens die Dimension 2 besitzt. Sollte dies nicht der Fall sein, wird der Algorithmus erst gar nicht ausgeführt.
  In den Zeilen 117 und 118 werden die beiden Attribute m_upperBound=Double.POSITIVE_INFINITY und m_shortestKnownTour=null gesetzt, um Initialwerte zu vergeben und die Vorbedingung von „branchAndBound()“ zu erfüllen.
  In Zeile 120 erfolgt dann der Aufruf des Branch-and-Bound-Algorithmus mit dem Startknoten 0, einem 0 Einheiten langen Anfangsstück und der Problem-Distanzmatrix. Nach dessen Termination ist eine minimale Rundreise in m_shortestKnownTour und die Länge dieser in m_upperBound gespeichert.

7. Komplexitäten

Es lassen sich Situationen konstruieren, wo alle (n-1)! Rundreisen konstruiert werden müssen, um die (in diesem Fall kann es nur eine sein) minimale Rundreise zu finden. Dies stellt den schlechtesten Fall dar, hier wächst der Aufwand in Abhängigkeit von der Anzahl n der Städte mit asymptotisch zur Fakultätsfunktion. Deshalb gilt: ***O(n!)***.

Im trivialen Fall der Aufzählung aller Permutationen ist der beste gleich dem schlechtesten Fall mit O(n!). Hier zeigt sich der Vorteil von Branch-and-Bound: der beste Fall lässt sich so konstruieren, dass die minimale Rundreise durch alle n Städte die Gestalt $0 \rightarrow 1 \rightarrow 2 \rightarrow \ldots \rightarrow n-1 \rightarrow 0$ hat und im Entscheidungsbaum nur diese eine Rundreise betrachtet werden muss. Alle Teilbäume, die diese Rundreise nicht enthalten, sollen aufgrund einer zu großen unteren Schranke abgeschnitten werden. Dadurch sind nur n Knoten im Entscheidungsbaum zu betrachten (inkl. der Wurzel), womit die Laufzeit im besten Fall linear ist, also gilt: $\Theta(n)$.

8. Quellennachweis

[1] http://www.inf.hs-zigr.de/~wagenkn/TI/Komplexitaet/ReferateSS02/ReferateSS97/TSP/tsp-html/tsp-Dateien/main.htm

[2] http://www.infosun.fmi.uni-passau.de/st/edu/pdp01/aufgabe3.pdf

[3] www-wi.uni-muenster.de/pi/lehre/SS03/Seminar/Suche_Christian_Hermanns.pdf