Datenstrukturen, Algorithmen und Programmierung II

Prof. Dr. Petra Mutzel
Markus Chimani
Carsten Gutwenger
Karsten Klein

Skript zur gleichnamigen Vorlesung von

Prof. Dr. Petra Mutzel

im Sommersemester 2008

VERSION 0.7
LEHRSTUHL 11 – ALGORITHM ENGINEERING
UNIVERSITÄT DORTMUND

Inhaltsverzeichnis

Kapitel 1

Optimierung

Optimierungsprobleme sind Probleme, die im Allgemeinen viele zulässige Lösungen besitzen. Jeder Lösung ist ein bestimmter Wert (Zielfunktionswert, Kosten) zugeordnet. Optimierungsalgorithmen suchen in der Menge aller zulässigen Lösungen diejenigen mit dem besten, dem *optimalen*, Wert. Beispiele für Optimierungsprobleme sind:

- Minimal aufspannender Baum (MST)
- Kürzeste Wege in Graphen
- Längste Wege in Graphen
- Handelsreisendenproblem (Traveling Salesman Problem, TSP)
- Rucksackproblem
- Scheduling (z.B. Maschinen, Crew)
- Zuschneideprobleme (z.B. Bilderrahmen, Anzüge)
- Packungsprobleme

Wir unterscheiden zwischen Optimierungsproblemen, für die wir Algorithmen mit polynomiellem Aufwand kennen („in P“), und solchen, für die noch kein polynomieller Algorithmus bekannt ist. Darunter fällt auch die Klasse der NP-schwierigen Optimierungsprobleme. Die Optimierungsprobleme dieser Klasse besitzen die Eigenschaft, dass das Finden eines polynomiellen Algorithmus für eines dieser Optimierungsprobleme auch polynomielle Algorithmen für alle anderen Optimierungsprobleme dieser Klasse nach sich ziehen würde. Die meisten Informatiker glauben, dass für NP-schwierige Optimierungsprobleme keine polynomiellen Algorithmen existieren. Die oben erwähnten Beispiele für Optimierungsprobleme

sind fast alle NP-schwierig, lediglich für das MST-Problem sowie das kürzeste Wege Probleme sind Algorithmen mit polynomieller Laufzeit bekannt.

Ein Algorithmus hat polynomielle Laufzeit, wenn die Laufzeitfunktion $T(n)$ durch ein Polynom in n beschränkt ist. Dabei steht n für die Eingabegröße der Instanz. Z.B. ist das Kürzeste-Wegeproblem polynomiell lösbar, während das Längste-Wegeproblem im Allgemeinen NP-schwierig ist. (Transformation vom Handlungsreisendenproblem: Wäre ein polynomieller Algorithmus für das Längste-Wegeproblem bekannt, dann würde dies direkt zu einem polynomiellen Algorithmus für das Handlungsreisendenproblem führen.) Dies scheint auf den ersten Blick ein Widerspruch zu sein: wenn man minimieren kann, dann sollte man nach einer Kostentransformation („mal -1“) auch maximieren können. Allerdings ist für das Kürzeste-Wegeproblem nur dann ein polynomieller Algorithmus bekannt, wenn die Instanz keine Kreise mit negativen Gesamtkosten enthält. Und genau das ist nach der Kostentransformation nicht mehr gewährleistet.

Wir konzentrieren uns hier meist auf *kombinatorische Optimierungsprobleme*, das sind solche, die auf eine endliche Grundmenge beschränkt sind.

Definition 1.1. Ein *kombinatorisches Optimierungsproblem* ist folgendermaßen definiert. Gegeben sind eine endliche Menge E (*Grundmenge*), eine Teilmenge I der Potenzmenge 2^E von E (*zulässige Mengen*), sowie eine Kostenfunktion $c : E \to K$. Gesucht ist eine Menge $I^* \in I$, so dass

$$c(I^*) = \sum_{e \in I^*} c(e)$$

so groß (klein) wie möglich ist.

Hat man ein Optimierungsproblem gegeben, so bezeichnet man eine Instantiierung dieses Problems mit speziellen Eingabedaten als *Probleminstanz*. Im Prinzip lassen sich Instanzen von kombinatorischen Optimierungsproblemen durch Enumeration aller möglichen zulässigen Lösungen in endlicher Zeit lösen. Allerdings dauert eine Enumeration i.A. viel zu lange (s. Kapitel 1.3).

Für polynomielle Optimierungsaufgaben existieren verschiedene Strategien zur Lösung. Oft sind das speziell für das jeweilige Problem entwickelte Algorithmen. Manchmal jedoch greifen auch allgemeine Strategien wie das *Greedy-Prinzip* oder die *dynamische Programmierung*, die wir in diesem Kapitel kennen lernen werden.

NP-schwierige Optimierungsprobleme treten in der Praxis häufiger auf. Für sie betrachten wir *exakte Verfahren*, d.h. solche, die immer optimale Lösungen liefern, aber wegen des exponentiell steigenden Zeitaufwands im Allgemeinen nur für kleinere Probleminstanzen verwendet werden können. Als exakte Verfahren studieren wir *Enumeration* (s. Kapitel 1.3), *Branch-and-Bound* (s. Kapitel 1.4) und *Dynamische Programmierung* (s. Kapitel 1.5) an

ausgewählten Beispielen. In der Praxis werden für NP-schwierige Optimierungsprobleme häufig *Heuristiken* verwendet, die meist die optimale Lösung nur annähern.

Wir beginnen dieses Kapitel mit der Einführung in verschiedene Optimierungsprobleme und einfachen konstruktiven Heuristiken (s. Kapitel 1.1). Wir werden sehen, dass es auch Heuristiken gibt, die eine Gütegarantie für die Lösung abgeben. Diese Algorithmen nennt man *approximative Algorithmen* (s. Kapitel 1.2). Am Ende des Kapitels werden wir allgemeine *Verbesserungsheuristiken* kennenlernen, deren Ziel es ist, eine gegebene Lösung durch lokale Änderungen zu verbessern (s. Kapitel 1.6).

1.1 Heuristiken

In diesem Abschnitt stellen wir typische NP-schwierige Optimierungsprobleme vor, an denen wir die verschiedenen Algorithmen zur Lösung von Optimierungsproblemen demonstrieren werden. Es handelt sich hierbei um kombinatorische Optimierungsprobleme.

1.1.1 Travelling Salesman Problem

Ein klassisches kombinatorisches Optimierungsproblem ist das *Travelling Salesman Problem* (TSP) (auch symmetrisches TSP, Rundreiseproblem bzw. Handlungsreisendenproblem genannt).

Travelling Saleman Problem (TSP)	
Gegeben:	vollständiger ungerichteter Graph $G = (V, E)$ mit Gewichtsfunktion $c : E \to \mathbb{R}$
Gesucht:	Tour T (Kreis, der jeden Knoten genau einmal enthält) von G mit minimalem Gewicht $c(T) = \sum_{e \in T} c(e)$

Das TSP ist ein NP-schwieriges Optimierungsproblem. Man kann sich leicht überlegen, dass es in einer TSP-Instanz mit $n = |V|$ Städten genau $(n-1)!/2$ viele verschiedene Touren gibt. Die Enumeration aller möglichen Touren ist also nur für sehr wenige Städte (kleines n) möglich. Bereits für $n = 12$ gibt es 19.958.400 verschiedene Touren. Für $n = 25$ sind es bereits ca. 10^{23} Touren und für $n = 60$ ca. 10^{80} Touren (zum Vergleich: die Anzahl der Atome im Weltall wird auf ca. 10^{80} geschätzt. Selbst mit einem „Super-Rechner“, der 40 Millionen Touren pro Sekunde enummerieren könnte, würde für 10^{80} Touren ca. 10^{64} Jahre benötigen.

Und trotzdem ist es möglich mit Hilfe von ausgeklügelten Branch-and-Bound Verfahren (s. Kapitel 1.4) TSP-Instanzen mit bis zu 85.900 Städten praktisch (beweisbar!) optimal zu

Eingabe: vollständiger ungerichteter Graph $G = (V, E)$ mit Kantenkosten c(e)
Ausgabe: zulässige Lösung für die gegebene TSP-Instanz

1: Beginne mit einer leeren Tour $T := \emptyset$
2: Beginne an einem Knoten $v := v_0$ ▷ Ende der Tour
3: **while** noch freie Knoten existieren **do**
4: Suche den nächsten freien (noch nicht besuchten) Knoten w zu v ▷ billigste Kante (v, w)
5: Addiere die Kante (v, w) zu T.
6: **end while**
7: Addiere die Kante (v, v_0)

Listing 1.1: Nearest-Neighbor (NN) Heuristik für das TSP.

lösen. Dies jedoch nur nach relativ langer Rechenzeit (s. www.tsp.gatech.edu bzw. Folien zur Vorlesung). Mit diesen Methoden kann man auch TSP-Instanzen bis zu einer Größe von ca. 300 Städten relativ schnell lösen.

Wir betrachten in diesem Abschnitt eine einfache Greedy-Heuristik für das TSP. *Greedy-Algorithmen* sind *„gierige“* Verfahren zur exakten oder approximativen Lösung von Optimierungsaufgaben, welche die Lösung iterativ aufbauen (z.B. durch schrittweise Hinzunahme von Objekten) und sich in jedem Schritt für die jeweils lokal beste Lösung entscheiden. Eine getroffene Entscheidung wird hierbei niemals wieder geändert. Daher sind Greedy-Verfahren bezogen auf die Laufzeit meist effizient, jedoch führen sie nur in seltenen Fällen (wie etwa für das kürzeste Wegeproblem oder das MST-Problem) zur optimalen Lösung.

Die Nearest-Neighbor Heuristik ist in Algorithmus 1.1 gegeben. Sie beginnt an einem Startknoten und addiert iterativ jeweils eine lokal beste Kantezum Ende der Tour hinzu. Das Problem bei diesem Vorgehen ist, dass das Verfahren sich selbst ab und zu in eine Sackgasse führt, so dass besonders teure (lange) Kanten benötigt werden, um wieder weiter zu kommen. Auch die letzte hinzugefügte Kante ist hier i.A. sehr teuer (s. Beispiel auf den Folien).

Wie gut ist nun die Nearest-Neighbor Heuristik (NN-Heuristik) im allgemeinen? Man kann zeigen, dass es Probleminstanzen gibt, für die sie beliebig schlechte Ergebnisse liefert. Eine solche Instanz ist z.B. in Beispiel 1.1 gegeben.

Beispiel 1.1. *Eine Worst-Case Instanz für die NN-Heuristik:* Die Knoten der TSP-Instanz seien nummeriert von 1 bis n und die Kantengewichte sind wie folgt: $c(i, i + 1) = 1$ für $i = 1, ..., n - 1$, $c(1, n) = M$, wobei M eine sehr große Zahl ist) und $c(i, j) = 2$ sonst.

Eine von der NN-Heuristik produzierte Lösung mit Startknoten 1 enthält die Kanten $(1, 2), (2, 3), ..., (n - 1, n)$ und $(n, 1)$. Der Wert dieser Lösung ist $(n - 1) + M$.

Eine optimale Tour ist z.B. gegeben durch die Kanten $(1, 2), (2, 3), ..., (n-3, n-2), (n-2, n), (n, n - 1)$ und $(n - 1, 1)$. Der optimale Wert ist also $(n - 2) + 4 = n + 2$.

Da M beliebig groß sein kann, kann die NN-Heuristik einen Lösungswert liefern, der beliebig weit vom optimalen Lösungswert entfernt ist (also beliebig schlecht ist).

1.1.2 Verschnitt- und Packungsprobleme

Ein anderes klassiches NP-schwieriges Optimierungsproblem ist die *Eindimensionale Verschnittoptimierung*.

Eindimensionale Verschnittoptimierung	
Gegeben:	Gegenstände $1, ..., N$ der Größe w_i und beliebig viele Rohlinge der GroßeK
Gesucht:	Finde die kleinste Anzahl von Rohlingen, aus denen alle Gegenstände geschnitten werden können.

Zu unserem Verschnittproblem gibt es ein äquivalentes Packungsproblem, das *Bin-Packing Problem*, bei dem es um möglichst gutes Packen von Kisten geht:

Bin-Packing Problem	
Gegeben:	Gegenstände $1, \ldots, N$ der Größe w_i und beliebig viele Kisten der Größe K.
Gesucht:	Finde die kleinste Anzahl von Kisten, die alle Gegenstände aufnehmen.

Eine bekannte Greedy-Heuristik ist die *First-Fit Heuristik* (FF-Heuristik), die folgendermaßen vorgeht: Die Gegenstände werden in einer festen Reihenfolge betrachtet. Jeder Gegenstand wird in die erstmögliche Kiste gelegt, in die er paßt. Beispiel 1.2 führt die FF-Heuristik an einer Probleminstanz durch.

Beispiel 1.2. Wir führen die First-Fit Heuristik an der folgenden Probleminstanz durch. Gegeben sind beliebig viele Kisten der Größe $K = 101$ und 37 Gegenstände der folgenden Größe: 7× Größe 6, 7× Größe 10, 3× Größe 16, 10× Größe 34, und 10× Größe 51.

Die First-Fit Heuristik berechnet die folgende Lösung:

1	×	(7×) 6 | (5×)10	$\sum = 92$
1	×	(2×)10 | (3×)16	$\sum = 68$
5	×	(2×)34	$\sum = 68$
10	×	(1×)51	$\sum = 51$

Das ergibt insgesamt 17 Kisten.

Wieder fragen wir uns, wie gut die von der First-Fit Heuristik berechneten Lösungen im allgemeinen sind. Für unser Beispiel sieht die optimale Lösung folgendermaßen aus:

$3\times$	51	34	16		$\sum = 101$
$7\times$	51	34	10	6	$\sum = 101$

Die dargestellte Lösung benötigt nur 10 Kisten. Offensichtlich ist diese auch die optimale Lösung, da jede Kiste auch tatsächlich ganz voll gepackt ist.

Für die Probleminstanz P_1 in Beispiel 1.2 weicht also der Lösungswert der First-Fit Heuristik $c_{FF}(P_1)$ nicht beliebig weit von der optimalen Lösung $c_{opt}(P_1)$ ab. Es gilt: $c_{FF}(P_1) \leq \frac{17}{10} c_{opt}(P_1)$. Aber vielleicht existiert ja eine andere Probleminstanz, bei der die FF-Heuristik nicht so gut abschneidet? Wir werden dieser Frage im nächsten Kapitel nachgehen.

1.1.3 0/1-Rucksackproblem

Wir betrachten ein weiteres NP-schwieriges Optimierungsproblem, nämlich das *0/1-Rucksackproblem*. Hier geht es darum aus einer gegebenen Menge von Gegenständen möglichst wertvolle auszuwählen, die in einem Rucksack mit gegebener Maximalgröße zu verpacken sind.

0/1-Rucksackproblem	
Gegeben:	N Gegenstände i mit Gewicht (Größe) w_i, und Wert (Kosten) c_i, und ein Rucksack der Größe K.
Gesucht:	Menge der in den Rucksack gepackten Gegenstände mit maximalem Gesamtwert; dabei darf das Gesamtgewicht den Wert K nicht überschreiten.

Wir führen 0/1-Entscheidungsvariablen für die Wahl der Gegenstände ein:

$$x_1, \ldots, x_N, \text{ wobei } x_i = \begin{cases} 0 & \text{falls Element } i \text{ nicht gewählt wird} \\ 1 & \text{falls Element } i \text{ gewählt wird} \end{cases}$$

Das 0/1-Rucksackproblem lässt sich nun formal folgendermaßen definieren:

Maximiere

$$\sum_{i=1}^{N} c_i x_i,$$

wobei

$$x_i \in \{0,1\} \wedge \sum_{i=1}^{N} w_i x_i \leq K.$$

Anmerkung 1.1. Das Rucksackproblem taucht in vielen Varianten auf, u.a. auch in einer Variante, bei der man mehrere Gegenstände des gleichen Typs hat. Um unser Problem von dieser Variante abzugrenzen, nennen wir unser Problem 0/1-Rucksackproblem. Wenn es jedoch aus dem Zusammenhang klar ist, welche Variante hier gemeint ist, dann verzichten wir auch auf den Zusatz 0/1.

Die *Greedy-Heuristik* für das 0/1-Rucksackproblem geht folgendermaßen vor. Zunächst werden die Gegenstände nach ihrem Nutzen sortiert. Der Nutzen von Gegenstand i ist gegeben durch c_i/w_i. Für alle Gegenstände i wird dann in der sortierten Reihenfolge ausprobiert, ob der Gegenstand noch in den Rucksack paßt. Wenn dies so ist, dann wird er hinzugefügt.

Beispiel 1.3. Das Rucksack-Problem mit $K = 17$ und folgenden Daten:

Gegenstand	a	b	c	d	e	f	g	h
Gewicht	3	4	4	6	6	8	8	9
Wert	3	5	5	10	10	11	11	13
Nutzen	1	1,25	1,25	1,66	1,66	1,375	1,375	1,44

Der Nutzen eines Gegenstands i errechnet sich aus c_i/w_i.
Das Packen der Gegenstände a, b, c und d führt zu einem Gesamtwert von 23 bei dem maximal zulässigen Gesamtgewicht von 17. Entscheidet man sich für die Gegenstände c, d und e, dann ist das Gewicht sogar nur 16 bei einem Gesamtwert von 25.

Die Greedy-Heuristik sortiert die Gegenstände zuerst nach ihrem Nutzen, damit erhalten wir die Reihenfolge d, e, h, f, g, b, c, a. Es werden also die Gegenstände d, e und b in den Rucksack gepackt mit Gesamtwert 25.

Wir stellen uns wieder die Frage, wie gut nun die Greedy-Heuristik im allgemeinen ist. Hierzu zeigt uns die in Beispiel 1.4 gegebene Instanz, dass die erzielten Lösungswerte der Greedy-Heuristik beliebig weit von den optimalen Werten entfernt sein können.

Beispiel 1.4. *Eine Worst-Case Rucksack-Probleminstanz für die Greedy-Heuristik.* Die gegebenen Gegenstände haben jeweils Gewichte und Werte $w_i = c_i = 1$ für $i = 1, ..., N-1$ und der n-te Gegenstand $c_n = K - 1$ und $w_N = K = MN$, wobei M eine sehr große Zahl ist. Der Rucksack hat Größe K (ist also sehr groß).

Die Greedy-Heuristik packt die ersten $N - 1$ Gegenstände in den Rucksack. Danach ist kein Platz mehr für das N-te Element. Die erzielte Lösung hat einen Lösungswert gleich $N - 1$.

Die optimale Lösung hingegen packt nur das N-te Element ein und erreicht dadurch einen Lösungswert von $K - 1 = MN - 1$.

Da M beliebig groß sein kann, ist auch dies eine Instanz bei der die Greedy-Heuristik beliebig *schlecht* sein kann, da der berechnete Lösungswert beliebig weit vom optimalen Wert entfernt sein kann.

Wir haben also in diesem Abschnitt drei verschiedene Greedy-Heuristiken für drei verschiedene NP-schwierige Optimieruungsprobleme kennengelernt. Dabei haben wir gesehen, dass manchmal die erzielte Lösung sehr weit von der Optimallösung entfernt sein kann. Im Fall der First-Fit Heuristik für Bin-Packing haben wir jedoch eine solche *schlechte* Instanz nicht gefunden. Im folgenden Abschnitt gehen wir diesen Effekten genauer nach.

1.2 Approximative Algorithmen und Gütegarantien

Ist ein Problem NP-schwierig, so können größere Instanzen im Allgemeinen nicht exakt gelöst werden, d.h. es besteht kaum Aussicht, mit Sicherheit optimale Lösungen in akzeptabler Zeit zu finden. In einer solchen Situation ist man dann auf *Heuristiken* angewiesen, also Verfahren, die zulässige Lösungen für das Optimierungsproblem finden, die aber nicht notwendigerweise optimal sind.

In diesem Abschnitt beschäftigen wir uns mit Heuristiken, die Lösungen ermitteln, über deren Qualität (Güte) man bestimmte Aussagen treffen kann. Man nennt solche Heuristiken mit Gütegarantien *approximative Algorithmen*.

Die „Güte“ eines Algorithmus sagt etwas über seine Fähigkeiten aus, optimale Lösungen gut oder schlecht anzunähern. Die formale Definition lautet wie folgt:

Definition 1.2. Sei A ein Algorithmus, der für jede Probleminstanz P eines Optimierungsproblems Π eine zulässige Lösung mit positivem Wert liefert. Dann definieren wir $c_A(P)$ als den Wert der Lösung des Algorithmus A für Probleminstanz $P \in \Pi$; $c_{\text{opt}}(P)$ sei der optimale Wert für P.

Für Minimierungsprobleme gilt: Falls

$$\frac{c_A(P)}{c_{\text{opt}}(P)} \leq \varepsilon$$

für alle Probleminstanzen P und $\varepsilon > 0$, dann heißt A ein ε-*approximativer* Algorithmus und die Zahl ε heißt *Gütegarantie* von Algorithmus A.

Für Maximierungsprobleme gilt: Falls

$$\frac{c_A(P)}{c_{\text{opt}}(P)} \geq \varepsilon$$

für alle Probleminstanzen P und $\varepsilon > 0$, dann heißt A ein ε-*approximativer* Algorithmus und die Zahl ε heißt *Gütegarantie* von Algorithmus A.

Beispiel 1.5. Wir betrachten als Beispiel ein Minimierungsproblem und jeweils die Lösungen einer Heuristik A und die optimale Lösung zu verschiedenen Probleminstanzen P_i:

$$c_A(P_1) = 100 \quad \text{und} \quad c_{\text{opt}}(P_1) = 50$$
$$c_A(P_2) = 70 \quad \text{und} \quad c_{\text{opt}}(P_2) = 40$$

Die Information, die wir daraus erhalten, ist lediglich, dass die Güte der Heuristik A nicht besser als 2 sein kann. Nur, falls

$$\frac{c_A(P_i)}{c_{\text{opt}}(P_i)} \leq 2$$

für alle möglichen Probleminstanzen P_i gilt, ist A ein 2-approximativer Algorithmus.

Anmerkung 1.2. Es gilt

- für Minimierungsprobleme: $\varepsilon \geq 1$,
- für Maximierungsprobleme: $\varepsilon \leq 1$,
- $\varepsilon = 1 \Leftrightarrow A$ ist exakter Algorithmus

Im Folgenden betrachten wir verschiedene approximative Algorithmen für das Bin-Packing-Problem und für das Travelling Salesman Problem.

1.2.1 Bin-Packing – Packen von Kisten

In Beispiel 1.2 hat die First-Fit Heuristik eine Lösung mit 17 Kisten generiert, während die optimale Lösung nur 10 Kisten benötigt. Daraus folgt

$$\frac{c_{FF}(P_1)}{c_{\text{opt}}(P_1)} = \frac{17}{10}$$

für diese Probleminstanz P_1. Dies lässt bisher nur den Rückschluss zu, dass die Gütegarantie der FF-Heuristik auf keinen Fall besser als $\frac{17}{10}$ sein kann.

Wir beweisen zunächst eine Güte von asymptotisch 2 für die First-Fit Heuristik. Das heißt also, dass die Lösungen der FF-Heuristik für alle möglichen Probleminstanzen nie mehr als doppelt so weit von der optimalen Lösung entfernt sein können. Die FF-Heuristik ist also ein Approximationsalgorithmus mit Faktor 2 für das Bin-Packing Problem.

Theorem 1.1. *Es gilt:* $c_{FF}(P) \leq 2c_{\text{opt}}(P) + 1$ *für alle* $P \in \Pi$.

Beweis. Offensichtlich gilt: Jede FF-Lösung füllt alle bis auf eine der belegten Kisten mindestens bis zur Hälfte. Daraus folgt für alle Probleminstanzen $P \in \Pi$:

$$c_{FF}(P) \leq \frac{\sum_{j=1}^{N} w_j}{K/2} + 1$$

Da

$$\sum_{j=1}^{N} w_j \leq K c_{\text{opt}}(P)$$

folgt

$$c_{FF}(P) \leq 2c_{\text{opt}}(P) + 1 \quad \Rightarrow \quad \frac{c_{FF}(P)}{c_{\text{opt}}(P)} \leq 2 + \frac{1}{c_{\text{opt}}(P)}$$

□

Man kann sogar noch eine schärfere Güte zeigen. Es gilt (ohne Beweis):

$$\frac{c_{FF}(P)}{c_{\text{opt}}(P)} < \frac{17}{10} + \frac{2}{c_{\text{opt}}(P)} \qquad \forall P \in \pi$$

Weiterhin kann man zeigen, dass $\frac{17}{10}$ der asymptotisch beste Approximationsfaktor für die FF-Heuristik ist. Für interessierte Studierende ist der Beweis in Johnson, Demers, Ullman, Garey, Graham in *SIAM Journal on Computing*, vol. 3 no. 4, S. 299-325, 1974, zu finden.

1.2.2 Das symmetrische TSP

Wir haben im vorigen Abschnitt gesehen, dass die Nearest-Neighbor Heuristik Lösungen für das TSP generieren kann, die beliebig weit von dem optimalen Lösungswert entfernt sein können. Im folgenden betrachten wir eine andere beliebte Heuristik für das TSP, die *Spanning Tree Heuristik*.

Eingabe: vollständiger ungerichteter Graph $G = (V, E)$ mit Kantenkosten c(e)
Ausgabe: zulässige Tour T für die gegebene TSP-Instanz

1: Bestimme einen minimalen aufspannenden Baum B von K_n.
2: Verdopple alle Kanten aus $B \rightarrow$ Graph (V, B_2).
3: Bestimme eine Eulertour F im Graphen (V, B_2).
4: Gib dieser Eulertour F eine Orientierung $\rightarrow F$
5: Wähle einen Knoten $i \in V$, markiere i, setze $p = i, T = \emptyset$.
6: **while** noch unmarkierte Knoten existieren **do**
7: Laufe von p entlang der Orientierung F bis ein unmarkierter Knoten q erreicht ist.
8: Setze $T = T \cup \{(p, q)\}$, markiere q, und setze $p = q$.
9: **end while**
10: Setze $T = T \cup \{(p, i)\} \rightarrow$ STOP; T ist die Ergebnis-Tour.

Listing 1.2: Spanning Tree (ST) Heuristik für das TSP.

Spanning-Tree-Heuristik (ST)

Die Spanning Tree (ST) Heuristik basiert auf der Idee einen minimal aufspannenden Baum zu generieren und daraus eine Tour abzuleiten. Dafür bedient sie sich einer sogenannten Eulertour.

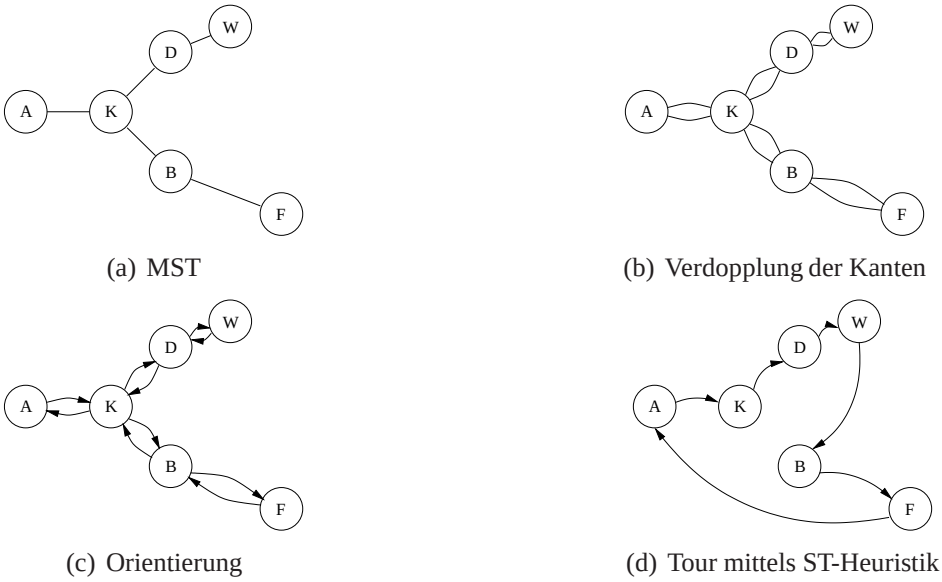
Definition 1.3. Eine *Eulertour* ist ein geschlossener Kantenzug, der jede Kante des Graphen genau einmal enthält.

In einem Graphen G existiert genau dann eine Eulertour wenn alle Knotengrade in G gerade sind. Dies kann man sich leicht überlegen: Jede Eulertour, die einen Knoten besucht (hineingeht) muss auch wieder aus dem Knoten heraus. Eine Eulertour kann in linearer Zeit berechnet werden (ohne Beweis).

Die ST-Heuristik berechnet also zuerst einen Minimum Spanning Tree T und verdoppelt dann die Kanten in T um alle Knotengrade gerade zu bekommen. Die Eulertour ist jedoch keine zulässige TSP-Tour, da diese im allgemeinen Knoten mehrfach besuchen kann. Deswegen muss aus der Eulertour eine zulässige Tour konstruiert werden. Dies wird durch "Abkürzungen" entlang der Eulertour erreicht. Listing 1.2 zeigt die einzelnen Schritte der ST-Heuristik.

Abbildung 1.1 zeigt eine Visualisierung der Spanning-Tree Heuristik anhand eines Beispiels (Städte in Nord-Rhein-Westfalen und Hessen: Aachen, Köln, Bonn, Düsseldorf, Frankfurt, Wuppertal).

Analyse der Gütegarantie. Leider ist das allgemeine TSP sehr wahrscheinlich nicht mit einer Konstanten approximierbar. Denn man kann zeigen: Gibt es ein $\varepsilon > 1$ und einen polyno-



(a) MST (b) Verdopplung der Kanten

(c) Orientierung (d) Tour mittels ST-Heuristik

Abbildung 1.1: ST-Heuristik

miellen Algorithmus A, der für jedes symmetrische TSP eine Tour T_A liefert mit $\frac{c_A(P)}{c_{\text{opt}}(P)} \leq \varepsilon$, dann ist P = NP.

Theorem 1.2. *Das Problem, das symmetrische TSP für beliebiges $\varepsilon > 1$ zu approximieren ist NP-schwierig (ohne Beweis).*

Doch es gibt auch positive Nachrichten. Wenn die gegebene TSP-Instanz gewisse Eigenschaften aufweist, dann ist diese doch sehr gut approximierbar:

Definition 1.4. Ein TSP heißt *euklidisch* wenn für die Distanzmatrix C die Dreiecks-Ungleichung gilt, d.h. für alle Knoten i, j, k gilt: $c_{ik} \leq c_{ij} + c_{jk}$ und alle $c_{ii} = 0$ sind.

Das heißt: es kann nie kürzer sein, von i nach k einen Umweg über eine andere Stadt j zu machen. Denkt man beispielsweise an Wegstrecken, so ist die geforderte Dreiecksungleichung meist erfüllt. Gute Nachrichten bringt das folgende Theorem:

Theorem 1.3. *Für die Probleminstanz P seien $c_{ST}(P)$ der von der ST-Heuristik erzielte Lösungswert und $c_{\text{opt}}(P)$ der Optimalwert. Für das euklidische TSP und die ST-Heuristik gilt:*

$$\frac{c_{ST}(P)}{c_{\text{opt}}(P)} \leq 2 \quad \text{für alle } P \in \Pi.$$

Ausgabe: Christofides-Tour: Ersetze Schritt 2: in Listing 1.2 (ST-Heuristik) durch:
1: Sei W die Menge der Knoten in (V, B) mit ungeradem Grad.
2: Bestimme im von W induzierten Untergraphen von K_n ein Minimum Perfektes Matching M.
3: Setze $B_2 = B \cup M$

Listing 1.3: Neue Teile der Christophides (CH) Heuristik für das TSP.

Beweis. Es gilt

$$c_{ST}(P) \leq c_{B_2}(P) = 2c_B(P) \leq 2c_{\text{opt}}(P).$$

Die erste Abschätzung gilt wegen der Dreiecksungleichung, die letzte, da ein Minimum Spanning Tree die billigste Möglichkeit darstellt, in einem Graphen alle Knoten zu verbinden. □

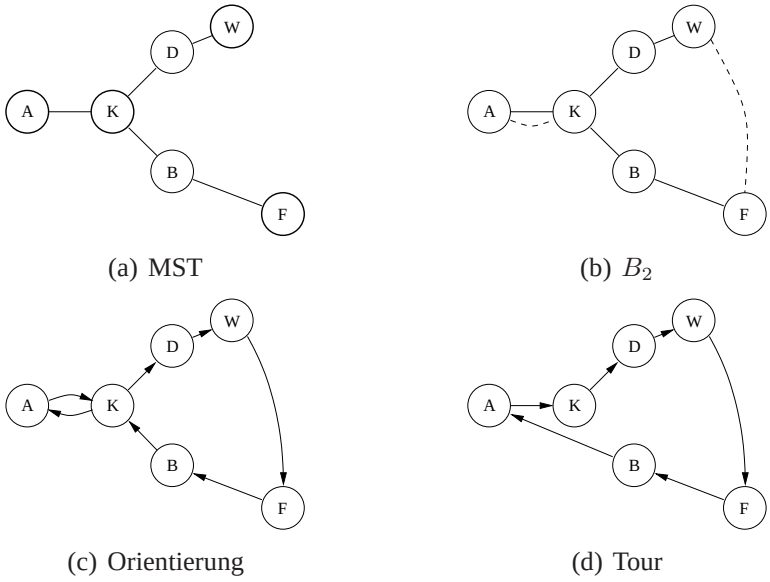
Im folgenden Abschnitt lernen wir eine Heuristik kennen, die das TSP noch besser approximiert.

Christophides-Heuristik (CH)

Diese 1976 publizierte Heuristik funktioniert ähnlich wie die Spanning-Tree-Heuristik, jedoch erspart man sich die Verdopplung der Kanten. Diese wird in der ST-Heuristik nur deswegen gemacht, weil dann sichergestellt ist, dass in dem Graphen eine Eulertour existiert. Es genügt jedoch statt einer Verdopplung aller Kanten, nur jeweils Kanten an den ungeraden Knoten zu dem Spanning Tree hinzuzunehmen. Diese zusätzliche Kantenmenge entspricht einem sogenannten Perfekten Matching auf den ungeraden Knoten im Spanning Tree.

Definition 1.5. Ein *Perfektes Matching* M in einem Graphen ist eine Kantenmenge, die jeden Knoten genau einmal enthält. Ein *Minimum Perfektes Matching* M ist ein Perfektes Matching mit dem kleinsten Gesamtgewicht, d.h. die Summe aller Kantengewichte c_e über die Kanten $e \in M$ ist minimal unter allen Perfekten Matchings.

Ein Perfektes Matching zwischen den ungeraden Knoten ordnet also jedem solchen Knoten einen eindeutigen Partnerknoten zu (Alle ungeraden Knoten werden zu Paaren zusammengefasst). Das Matching „geht auf“, denn wir wissen: die Anzahl der ungeraden Knoten ist immer gerade (s. Abschnitt **??**). Um eine gute Approximationsgüte zu erreichen, berechnet die Christophides-Heuristik nicht irgendein Perfektes Matching, sondern dasjenige mit kleinstem Gesamtgewicht. Ein Minimum Perfektes Matching kann in polynomieller Zeit berechnet werden (ohne Beweis). Listing 1.3 zeigt die Schritte der Christofides-Heuristik, die Schritt (2) aus dem Listing 1.2 der ST-Heuristik ersetzen.



(a) MST (b) B_2

(c) Orientierung (d) Tour

Abbildung 1.2: Die CH-Heuristik an einem Beispiel

Abbildung 1.2 zeigt die Konstruktion an unserem Beispiel. Dabei betrachten wir für die Berechnung eines Minimalen Perfekten Matchings den vollständigen Untergraphen, der von den Knoten A, K, W, F induziert wird.

Analyse der Gütegarantie. Die Christofides-Heuristik erreicht eine bessere Approximationsgüte als die Spanning-Tree Heuristik für das euklidische TSP.

Theorem 1.4. *Für die Probleminstanz P seien $c_{CH}(P)$ der von der CH-Heuristik erzielte Lösungswert und $c_{\text{opt}}(P)$ der Optimalwert. Für das euklidische TSP und die Christophides-Heuristik gilt:*

$$\frac{c_{CH}(P)}{c_{\text{opt}}(P)} \leq \frac{3}{2} \text{ für alle } P \in \Pi$$

Beweis. Seien $i_1, i_2, \ldots, i_{2M}$ die Knoten von B mit ungeradem Grad und zwar so nummeriert, wie sie in einer optimalen Tour T_{opt} vorkommen, d.h. $T_{\text{opt}} = \{i_1, \ldots, i_2, \ldots, i_{2M}, \ldots\}$. Sei $M_1 = \{(i_1, i_2), (i_3, i_4), \ldots\}$ und $M_2 = \{(i_2, i_3), \ldots, (i_{2M}, i_1)\}$. Es gilt:

$$c_{\text{opt}}(P) \geq c_{M_1}(P) + c_{M_2}(P) \geq c_M(P) + c_M(P)$$

Die erste Abschätzung gilt wegen der Dreiecks-Ungleichung (Euklidisches TSP). Die zweite Abschätzung gilt, weil M das Matching kleinsten Gewichts ist. Weiterhin gilt:

$$c_{CH}(P) \leq c_{B_2}(P) = c_B(P) + c_M(P) \leq c_{\text{opt}}(P) + \frac{1}{2}c_{\text{opt}}(P) = \frac{3}{2}c_{\text{opt}}(P).$$

□

Anmerkung 1.3. Die Christophides-Heuristik (1976) war lange Zeit die Heuristik mit der besten Gütegarantie. Erst im Jahr 1996 zeigte Arora, dass das euklidische TSP beliebig nah approximiert werden kann: die Gütegarantie $\varepsilon > 1$ kann mit Laufzeit $O(N^{\frac{1}{\varepsilon-1}})$ approximiert werden. Eine solche Familie von Approximationsalgorithmen wird als *Polynomial Time Approximation Scheme* (PTAS) bezeichnet.

Anmerkung 1.4. Konstruktionsheuristiken für das symmetrische TSP erreichen in der Praxis meistens eine Güte von ca. 10-15% Abweichung von der optimalen Lösung. Die Christophides-Heuristik liegt bei ca. 14%.

1.3 Enumerationsverfahren

Exakte Verfahren, die auf einer vollständigen Enumeration beruhen, eignen sich für Optimierungsprobleme diskreter Natur. Für solche kombinatorische Optimierungsprobleme ist es immer möglich, die Menge aller zulässigen Lösungen aufzuzählen und mit der Kostenfunktion zu bewerten. Die am besten bewertete Lösung ist die optimale Lösung. Bei NP-schwierigen Problemen ist die Laufzeit dieses Verfahrens dann nicht durch ein Polynom in der Eingabegröße beschränkt.

Im Folgenden betrachten wir ein Enumerationsverfahren für das 0/1-Rucksackproblem.

1.3.1 Ein Enumerationsalgorithmus für das 0/1-Rucksack-Problem

Eine Enumeration aller zulässigen Lösungen für das 0/1-Rucksackproblem entspricht der Aufzählung aller Teilmengen einer N-elementigen Menge (bis auf diejenigen Teilmengen, die nicht in den Rucksack passen).

Der Algorithmus *Enum()* (s. Listing 1.4) basiert auf genau dieser Idee. Zu jedem Lösungsvektor x gehört ein Zielfunktionswert (Gesamtkosten von x) $xcost$ und ein Gesamtgewichtswert $xweight$. Die bisher beste gefundene Lösung wird in dem globalen Vektor $bestx$ und der zugehörige Lösungswert in der globalen Variablen $maxcost$ gespeichert. Der Algorithmus wird mit dem Aufruf $Enum(0, 0, 0, x)$ gestartet.

In jedem rekursiven Aufruf wird die aktuelle Lösung x bewertet. Danach werden die Variablen $x[1]$ bis $x[z]$ als fixiert betrachtet. Der dadurch beschriebene Teil des gesamten Suchraums wird weiter unterteilt: Wir betrachten alle möglichen Fälle, welche Variable $x[i]$ mit $i = z+1$ bis $i = N$ als nächstes auf 1 gesetzt werden kann; die Variablen $x[z+1]$ bis $x[i-1]$ werden gleichzeitig auf 0 fixiert. Alle so erzeugten kleineren Unterprobleme werden durch rekursive Aufrufe gelöst.

Wir folgen hier wieder dem Prinzip des *Divide & Conquer* („Teile und herrsche“), wie wir es beispielsweise schon von Sortieralgorithmen kennen: Das Problem wird rekursiv in kleinere Unterprobleme zerteilt, bis die Unterprobleme trivial gelöst werden können.

Eingabe: Anzahl z der fixierten Variablen in x; Gesamtkosten $xcost$; Gesamtgewicht $xweight$; aktueller Lösungsvektor x
Ausgabe: aktualisiert die globale bisher beste Lösung $bestx$ und ihre Kosten $maxcost$, wenn eine bessere Lösung gefunden wird

```
1:  if xweight ≤ K then
2:      if xcost > maxcost then
3:          maxcost = xcost;
4:          bestx = x;
5:      end if
6:      for i = z + 1, ..., N do
7:          x[i] = 1;
8:          Enum (i, xcost + c[i], xweight + w[i], x);
9:          x[i] = 0;
10:     end for
11: end if
```

Listing 1.4: Enum $(z, xcost, xweight, x)$;

Der Algorithmus ist korrekt, denn die Zeilen (6)-(9) enumerieren über alle möglichen Teilmengen einer N-elementigen Menge, die in den Rucksack passen. Die Zeilen (1)-(5) sorgen dafür, dass nur zulässige Lösungen betrachtet werden und die optimale Lösung gefunden wird.

Analyse der Laufzeit. Offensichtlich ist die Laufzeit in $O(2^N)$.

Wegen der exponentiellen Laufzeit ist das Enumerationsverfahren i.A. nur für kleine Instanzen des 0/1-Rucksackproblems geeignet. Bereits für $N \geq 50$ ist das Verfahren nicht mehr praktikabel. Deutlich besser geeignet für die Praxis ist das Verfahren der Dynamischen Programmierung, das wir in Abschnitt 1.5.1 betrachten werden.

1.4 Branch-and-Bound

Eine spezielle Form der beschränkten Enumeration, die auch auf dem Divide-&-Conquer-Prinzip basiert, ist das Branch-and-Bound Verfahren. Dabei werden durch Benutzung von Primal- und Dualheuristiken möglichst große Gruppen von Lösungen als nicht optimal bzw. gültig erkannt und von der vollständigen Enumeration ausgeschlossen. Hierzu bedient man sich sogenannter unterer bzw. oberer Schranken.

Definition 1.6. Für eine Instanz P eines Optimierungsproblems heißt $L > 0$ *untere Schranke* (*lower bound*), wenn für den optimalen Lösungswert gilt $c_{\text{opt}}(P) \geq L$. Der Wert $U > 0$ heißt *obere Schranke* (*upper bound*), wenn gilt $c_{\text{opt}}(P) \leq U$.

Die Idee von Branch-and-Bound ist einfach. Wir gehen hier von einem Problem aus, bei dem eine Zielfunktion minimiert werden soll. (Ein Maximierungsproblem kann durch Vorzeichenumkehr in ein Minimierungsproblem überführt werden.)

- Zunächst berechnen wir z.B. mit einer Heuristik eine zulässige (im Allgemeinen nicht optimale) Startlösung mit Wert U und eine untere Schranke L für alle möglichen Lösungswerte (Verfahren hierzu nennt man auch *Dualheuristiken*).
- Falls $U = L$ sein sollte, ist man fertig, denn die gefundene Lösung muss optimal sein.
- Ansonsten wird die Lösungsmenge partitioniert (*Branching*) und die Heuristiken werden auf die Teilmengen angewandt (*Bounding*).
- Ist für eine (oder mehrere) dieser Teilmengen die für sie berechnete untere Schranke L (*lower bound*) nicht kleiner als die beste überhaupt bisher gefundene obere Schranke (*upper bound* = eine Lösung des Problems), braucht man die Lösungen in dieser Teilmenge nicht mehr beachten. Man erspart sich also die weitere Enumeration.
- Ist die untere Schranke kleiner als die beste gegenwärtige obere Schranke, muss man die Teilmengen weiter zerkleinern. Man fährt solange mit der Zerteilung fort, bis für alle Lösungsteilmengen die untere Schranke mindestens so groß ist wie die (global) beste obere Schranke.

Ein Branch-and-Bound Algorithmus besteht also aus den Schritten

- **Branching:** Partitioniere die Lösungsmenge
- **Search:** Wähle eines der bisher erzuegten Teilprobleme
- **Bounding:** Berechne für die ausgewählte Teilmenge je eine untere und obere Schranke und prüfe, ob dieses Teilproblem weiter betrachtet werden muß.

Listing 1.5 beschreibt dieses Grundprinzip im Pseudocode. Die Hauptschwierigkeit besteht darin, *gute* Zerlegungstechniken und einfache Datenstrukturen zu finden, die eine effiziente Abarbeitung und Durchsuchung der einzelnen Teilmengen ermöglichen. Außerdem sollten die Heuristiken möglichst gute Schranken liefern, damit möglichst große Teilprobleme ausgeschlossen werden können.

1.4.1 Branch-and-Bound für das 0/1-Rucksackproblem

In diesem Abschnitt stellen wir einen konkreten Branch-and-Bound Algorithmus für das 0/1-Rucksackproblem vor. Hierbei handelt es sich um ein Maximierungsproblem, d.h. jede zulässige Lösung bildet eine untere Schranke für die gegebene Probleminstanz P.

Eingabe: Minimierungsproblem P
Ausgabe: Optimale Lösung U (und globale obere Schranke)

```
1:  var Liste offener (Sub-)probleme Π; lokale untere Schranke L'; lokale heuristische
    Lösung U'
2:  Setze U = ∞; Π = {P};
3:  while ∃P' ∈ Π do
4:      entferne P' von Π
5:      berechne für P' lokale untere Schranke L' mit Dualheuristik;        ▷ Bounding:
6:      if L' < U then                ▷ Fall L' ≥ U braucht nicht weiter verfolgt zu werden
7:          berechne zulässige Lösung für P' mit Heuristik → obere Schranke U';
8:          if U' < U then
9:              U = U';                                  ▷ neue bisher beste Lösung gefunden
10:             entferne aus Π alle Subprobleme mit lokaler unterer Schranke ≥ U
11:         end if
12:         if L' < U then            ▷ Fall L' ≥ U braucht nicht weiter verfolgt zu werden
13:             partitioniere P' in Teilprobleme P_1, ..., P_k              ▷ Branching:
14:             Π = Π ∪ {P_1, ..., P_k};
15:         end if
16:     end if
17: end while
```

Listing 1.5: Branch-and-Bound für Minimierung

- **Init.** Eine zulässige Startlösung kann z.B. mit der Greedy-Heuristik berechnet werden. Diese bildet eine untere Schranke für P. Eine mögliche Berechnung einer oberen Schranke U wird weiter unten vorgestellt.

- **Branching.** Wir zerlegen das Problem in zwei Teilprobleme: Wir wählen Gegenstand i und wählen es für Teilproblem T_{1i} aus (d.h. $x_i = 1$) und für Teilproblem T_{i2} explizit nicht aus (d.h. $x_i = 0$). Für T_1 muss das Gewicht von Gegenstand i von der Rucksackgröße abgezogen werden und der Wert zum Gesamtwert hinzuaddiert werden. Danach streichen wir Gegenstand i aus unserer Liste.

- **Search.** Wähle eines der bisher erzeugten Teilprobleme, z.B. dasjenige mit der besten (größten) oberen Schranke, denn wir hoffen, dass wir hier die beste Lösung finden werden.

- **Bounding.** Berechne für eine dieser Teilmengen T_i je eine untere und obere Schranke L_i und U_i. Sei L der Wert der besten bisher gefundenen Lösung. Falls $U_i \leq L_i$, braucht man die Teillösungen in der Teilmenge T_i nicht weiter betrachten.

Eine obere Schranke kann z.B. mit dem folgenden Verfahren berechnet werden. Zunächst werden die Gegenstände nach ihrem Nutzen $f_i = c_i/w_i$ sortiert. Seien $g_1, g_2, \ldots, g_n$ die in dieser Reihenfolge sortierten Gegenstände. Berechne das maximale r mit $g_1 + g_2 + \cdots g_r \leq$

K. Genau diese r Gegenstände werden dann eingepackt (also $x_i = 1$ für $i = 1, \ldots, r$). Danach ist noch Platz für $K - (g_1 + \ldots + g_r)$ Einheiten an Gegenständen. Diesen freien Platz füllen wir mit $(K - (g_1 + \ldots + g_r))/g_{r+1}$ Einheiten von Gegenstand $r + 1$ auf.

Wir zeigen, dass der so berechnete Wert eine obere Schranke für die Probleminstanz darstellt. Denn die ersten r Gegenstände mit dem höchsten Nutzen pro Einheit sind im Rucksack enthalten. Und zwar jeweils so viel davon, wie möglich ist. Allerdings wird eventuell vom letzten Gegenstand $r+1$ ein nicht-gazzahliger Anteil eingepackt. Deswegen ist die generierte Lösung i.A. nicht zulässig (sie erfüllt die Ganzzahligkeitsbedingung i.A. nicht). Aber es gilt: der berechnete Lösungswert der Gegenstände im Rucksack ist mindestens so groß wie die beste Lösung.

Beispiel 1.6 zeigt die Berechnung einer oberen Schranke für die 0/1-Rucksack-Probleminstanz aus Beispiel 1.4.

Beispiel 1.6. Wir berechnen eine obere Schranke für Beispiel 1.4. Die Sortierung nach Nutzen ergibt die Reihenfolge d, e, h, f, g, b, c, a. Wir packen also die Gegenstände d und e in den Rucksack, d.h. $x_d = x_e = 1$. Danach ist noch Platz frei für 5 Einheiten, aber Gegenstand h hat leider Gewicht 9 und paßt nicht mehr ganz in den Rucksack hinein. Wir nehmen also noch $x_h = 5/9$ Einheiten dazu. Der Wert der aktuellen (nicht-zulässingen) Lösung ist $10 + 10 + 5/9(13) < 27,3$. Eine obere Schranke für die beste erreichbare Lösung der Probleminstanz ist also 27.

Die Berechnung einer oberen Schranke inmitten eines Branch-and-Bound Baumes (BB-Baumes), bei dem eine Teillösung bereits fixiert wurde, ist sehr ähnlich. Die bisher erhaltene Teillösung wird sukzessive um die noch nicht fixierten Gegenstände mit dem größten Nutzen erweitert. Dabei muss beachtet werden, dass bei jeder Fixierung eines Gegenstandes im BB-Baum die verbleibende Rucksackgröße neu berechnet wird bzw. die explizit nicht gewählten Gegenstände aus der iste gestrichen werden.

1.4.2 Branch-and-Bound für das asymmetrische TSP

Der „Großvater“ aller Branch-and-Bound Algorithmen in der kombinatorischen Optimierung ist das Verfahren von Little, Murty, Sweeny und Karel zur Lösung asymmetrischer Travelling-Salesman-Probleme, das 1963 veröffentlicht wurde.

Asymmetrisches Travelling Salesman Problem (ATSP)

Gegeben: Gerichteter vollständiger Graph $G(V, E)$ mit $N = |V|$ Knoten, die Städten entsprechen, und eine Distanzmatrix C mit $c_{ii} = +\infty$ und $c_{ij} \geq 0$.

Gesucht: Rundtour $T \subset E$ (Hamiltonscher Kreis) durch alle N Städte, die jede Stadt genau einmal besucht und minimalen Distanzwert aufweist:

$$c(T) = \sum_{(i,j)\in T} c_{ij}$$

Lösungsidee: Zeilen- und Spaltenreduktion der Matrix C und sukzessive Erzeugung von Teilproblemen beschrieben durch $P(EINS, NULL, I, J, C, L, U)$.

Dabei bedeutet:

$EINS$: Menge der bisher auf 1 fixierten Kanten
$NULL$: Menge der bisher auf 0 fixierten Kanten
I: Noch relevante Zeilenindizes von C
J: Noch relevante Spaltenindizes von C
C: Distanzmatrix des Teilproblems
L: Untere Schranke für das Teilproblem
U: Obere Schranke für das globale Problem

Vorgangsweise:

1. Das Anfangsproblem ist

   $$P(\emptyset, \emptyset, \{1, \ldots, n\}, \{1, \ldots, n\}, C, 0, \infty),$$

   wobei C die Ausgangsmatrix ist.
   Setze dieses Problem auf eine globale Liste der zu lösenden Probleme.

2. Sind alle Teilprobleme gelöst $\longrightarrow$ STOP.

   Andernfalls wähle ein ungelöstes Teilproblem

   $$P(EINS, NULL, I, J, C, L, U)$$

   aus der Problemliste.

3. **Bounding:**

   a) **Zeilenreduktion:**
   Für alle $i \in I$:

   Berechne das Minimum der i-ten Zeile von C

   $$c_{ij_0} = \min_{j\in J} c_{ij}$$

   Setze $c_{ij} = c_{ij} - c_{ij_0}\ \forall j \in J$ und $L = L + c_{ij_0}$

b) **Spaltenreduktion:**
Für alle $j \in J$:

Berechne das Minimum der j-ten Spalte von C

$$c_{i_0 j} = \min_{i \in I} c_{ij}$$

Setze $c_{ij} = c_{ij} - c_{i_0 j}\ \forall i \in I$ und $L = L + c_{i_0 j}$

c) Ist $L \geq U$, so entferne das gegenwärtige Teilproblem aus der Problemliste und gehe zu 2.

Die Schritte (d) und (e) dienen dazu, eine neue (globale) Lösung des Gesamtproblems zu finden. Dabei verwenden wir die Information der in diesem Teilproblem bereits festgesetzten Entscheidungen.

d) Definiere den „Nulldigraphen“ $G_0 = (V, A)$ mit
$A = EINS \cup \{(i,j) \in I \times J \mid c_{ij} == 0\}$

e) Versuche, mittels einer Heuristik, eine Rundtour in G_0 zu finden. Wurde keine Tour gefunden, so gehe zu 4: Branching.
Sonst hat die gefundene Tour die Länge L. In diesem Fall:

e_1) Enferne alle Teilprobleme aus der Problemliste, deren lokale, untere Schranke größer gleich L ist.

e_2) Setze in allen noch nicht gelösten Teilproblemen $U = L$.

e_3) Gehe zu 2.

4. **Branching:**

a) Wähle nach einem Plausibilitätskriterium eine Kante $(i,j) \in I \times J$, die 0 bzw. 1 gesetzt wird.

b) Definiere die neuen Teilprobleme

b_1) $P(EINS \cup \{(i,j)\}, NULL, I \backslash \{i\}, J \backslash \{j\}, C', L, U)$
wobei C' aus C durch Streichen der Zeile i und der Spalte j entsteht – von i darf nicht noch einmal hinaus- und nach j nicht noch einmal hineingelaufen werden.

b_2) $P(EINS, NULL \cup \{(i,j)\}, I, J, C'', L, U)$,
wobei C'' aus C entsteht durch $c_{ij} = \infty$.

Füge diese Teilprobleme zur Problemliste hinzu und gehe zu 2.

Bemerkungen zum Algorithmus

zu 3. Bounding:

- Hinter der Berechnung des Zeilenminimums steckt die folgende Überlegung: Jede Stadt muss in einer Tour genau einmal verlassen werden. Zeile i der Matrix C enthält jeweils die Kosten für das Verlassen von Stadt i. Und das geht auf keinen Fall billiger als durch das Zeilenminimum c_{ij_0}. Das gleiche Argument gilt für das Spaltenminimum, denn jede Stadt muss genau einmal betreten werden, und die Kosten dafür sind in Spalte j gegeben.

- Schritte (a) und (b) dienen nur dazu, eine untere Schranke zu berechnen (Bounding des Teilproblems). Offensichtlich erhält man eine untere Schranke dieses Teilproblems durch das Aufsummieren des Zeilen- und Spaltenminimums.

- Für die Berechnung einer globalen oberen Schranke können Sie statt (d) und (e) auch jede beliebige TSP-Heuristik verwenden.

zu 4. Branching:

- Ein Plausibilitätskriterium ist beispielsweise das folgende:

  Seien

  $$u(i,j) = \min\{c_{ik} \mid k \in J \setminus \{j\}\} + \min\{c_{kj} \mid k \in I \setminus \{i\}\} - c_{ij}$$

  die minimalen Zusatzkosten, die entstehen, wenn wir von i nach j gehen, ohne die Kante (i,j) zu benutzen. Wir wählen eine Kante $(i,j) \in I \times J$, so dass

  $$c_{ij} = 0 \qquad \text{und} \qquad u(i,j) = \max\{u(p,q) \mid c_{pq} = 0\}.$$

- Dahinter steckt die Überlegung: Wenn die Zusatzkosten sehr hoch sind, sollten wir lieber direkt von i nach j gehen. Damit werden „gute“ Kanten beim Enumerieren erst einmal bevorzugt. Wichtig dabei ist, dass $c_{ij} = 0$ sein muss, sonst stimmen die Berechnungen von L und U nicht mehr.

- Bei der Definition eines neuen Teilproblems kann es vorkommen, dass die dort (in $EINS$ oder $NULL$) fixierten Kanten zu keiner zulässigen Lösung mehr führen können (weil z.B. die zu $EINS$ fixierten Kanten bereits einen Kurzzyklus enthalten). Es ist sicher von Vorteil, den Algorithmus so zu erweitern, dass solche Teilprobleme erst gar nicht zu der Problemliste hinzugefügt werden (denn in diesem Zweig des Enumerationsbaumes kann sich keine zulässige Lösung mehr befinden).

Durchführung des Branch & Bound Algorithmus an einem ATSP Beispiel

Die Instanz des Problems ist gegeben durch die Distanzmatrix C:

$$C = \begin{pmatrix} \infty & 5 & 1 & 2 & 1 & 6 \\ 6 & \infty & 6 & 3 & 7 & 2 \\ 1 & 4 & \infty & 1 & 2 & 5 \\ 4 & 3 & 3 & \infty & 5 & 4 \\ 1 & 5 & 1 & 2 & \infty & 5 \\ 6 & 2 & 6 & 4 & 5 & \infty \end{pmatrix}$$

Das Ausgangsproblem:

$$P = (\emptyset, \emptyset, \{1, \ldots, 6\}, \{1, \ldots, 6\}, C, 0, +\infty)$$

Bounding:

Die Zeilenreduktion ergibt:

$$C = \begin{pmatrix} \infty & 4 & 0 & 1 & 0 & 5 \\ 4 & \infty & 4 & 1 & 5 & 0 \\ 0 & 3 & \infty & 0 & 1 & 4 \\ 1 & 0 & 0 & \infty & 2 & 1 \\ 0 & 4 & 0 & 1 & \infty & 4 \\ 4 & 0 & 4 & 2 & 3 & \infty \end{pmatrix}$$

$$L = 1 + 2 + 1 + 3 + 1 + 2 = 10$$

Die Spaltenreduktion ändert hier nichts, da bereits in jeder Spalte das Minimum 0 beträgt.

Nun wird der Nulldigraph aufgestellt (siehe Abbildung 1.3). In diesem Nulldigraphen existiert keine Tour. 6 kann nur über 2 erreicht werden und von 6 müßte wieder auf 2 gegangen werden.

Branching: Wähle Kante (2,6) und setze diese einmal gleich 1 und einmal gleich 0. Dies ergibt zwei neue Teilprobleme.

Das erste neue Teilproblem:

$$P = (\{(2,6)\}, \emptyset, \{1, 3, \ldots, 6\}, \{1, \ldots, 5\}, C', 10, +\infty)$$

$$C' = \begin{pmatrix} \infty & 4 & 0 & 1 & 0 \\ 0 & 3 & \infty & 0 & 1 \\ 1 & 0 & 0 & \infty & 2 \\ 0 & 4 & 0 & 1 & \infty \\ 4 & \infty & 4 & 2 & 3 \end{pmatrix}$$

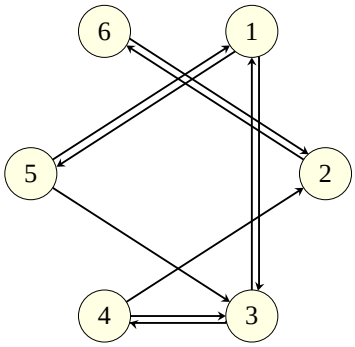


Abbildung 1.3: Der Nulldigraph in Branch & Bound

Die untere Schranke für dieses Teilproblem nach Zeilen- und Spaltenreduktion ist $L = 12$, da nur die Zeile mit Index (6) um 2 reduziert werden muss.

Das zweite neue Teilproblem:

$$P = (\emptyset, \{(2,6)\}, \{1,2,3,4,5,6\}, \{1,2,3,4,5,6\}, C'', 10, +\infty)$$

C'' entsteht aus C nach Setzen von C_{26} auf $+\infty$. Die untere Schranke für dieses Problem ist ebenfalls 12.

Abbildung 1.4 zeigt einen möglichen Enumerationsbaum für dieses Beispiel. (Dabei wird nicht das vorgestellte Plausibilitätskriterium genommen.) Hier wird als nächstes die Kante $(4,2)$ im Branching betrachtet, was Teilprobleme (4) und (5) erzeugt. Danach wird die Kante $(3,4)$ fixiert, was die Teilprobleme (6) und (7) begründet. Nach dem Fixieren der Kante $(5,1)$ wird zum ersten Mal eine globale obere Schranke von 13 gefunden (d.h. eine Tour im Nulldigraphen). Nun können alle Teilprobleme, deren untere Schranke gleich 13 ist, aus der Problemliste entfernt werden. In unserem Beispiel wird als nächstes das Teilproblem (3) gewählt, wobei dann über die Kante $(2,4)$ gebrancht wird, was Teilprobleme (9) und (10) erzeugt. Weil wir mit Teilproblem (10) niemals eine bessere Lösung als 15 erhalten können (untere Schranke), können wir diesen Teilbaum beenden. Wir machen mit Teilproblem (9) weiter und erzeugen die Probleme (11) und (12), die auch sofort beendet werden können. Es bleibt Teilproblem (7), das jedoch auch mit dem Branchen auf der Kante $(3,1)$ zu „toten“ Teilproblemen führt.

Analyse der Laufzeit. Die Laufzeit ist im Worst-Case exponentiell, denn jeder Branching-Schritt verdoppelt die Anzahl der neuen Teilprobleme. Dies ergibt im schlimmsten Fall bei N Städten eine Laufzeit von 2^N. Allerdings greift das Bounding im Allgemeinen recht gut, so dass man deutlich weniger Teilprobleme erhält. Doch bereits das Berechnen einer Instanz mit $N = 80$ Städten mit Hilfe dieses Branch-and-Bound Verfahrens dauert schon zu lange. Das vorgestellte Verfahren ist für TSPs mit bis zu 50 Städten anwendbar.

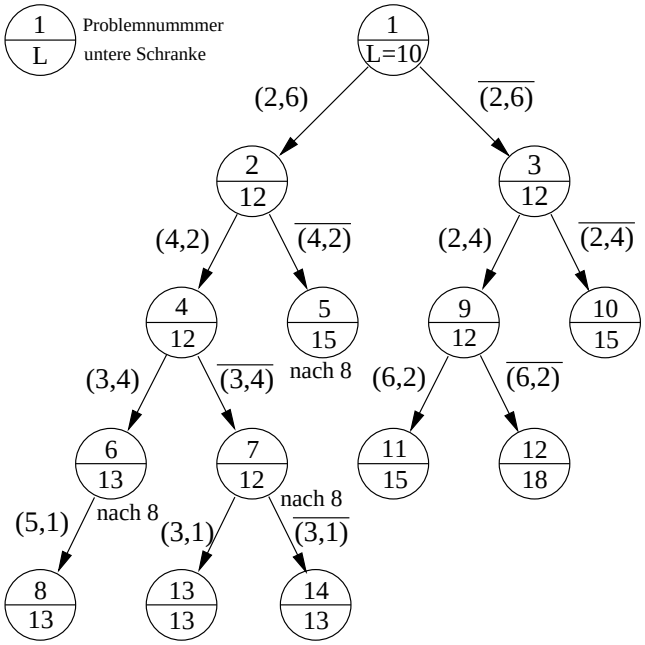


Abbildung 1.4: Möglicher Enumerationsbaum für das Beispiel

In der Praxis hat sich für die exakte Lösung von TSPs die Kombination von Branch-and-Bound Methoden mit linearer Programmierung bewährt. Solche Verfahren, die als Branch-and-Cut Verfahren bezeichnet werden, sind heute in der Lage in relativ kurzer Zeit TSPs mit ca. 500 Städten exakt zu lösen. Die größten nicht-trivialen bisher exakt gelösten TSPs wurden mit Hilfe von Branch-and-Cut Verfahren gelöst und umfassen ca. 15.000 Städte.

Typische Anwendungen von Branch-and-Bound Verfahren liegen in Brettspielen, wie z.B. Schach oder Springer-Probleme. Die dort erfolgreichsten intelligenten Branch-and-Bound Verfahren funktionieren in Kombination mit parallelen Berechnungen.

1.5 Dynamische Programmierung

Idee: Zerlege das Problem in kleinere Teilprobleme P_i ähnlich wie bei Divide & Conquer. Allerdings: während die P_i bei Divide & Conquer unabhängig sind, sind sie hier voneinander abhängig.

Dazu: Löse jedes Teilproblem und speichere das Ergebnis E_i so ab, dass E_i zur Lösung größerer Probleme verwendet werden kann.

Allgemeines Vorgehen:

1. Wie ist das Problem zerlegbar? Definiere den Wert einer optimalen Lösung rekursiv.

2. Bestimme den Wert der optimalen Lösung „bottom-up“.

Wir haben bereits einen Dynamischen Programmierungsalgorithmus kennengelernt: den Algorithmus von Floyd-Warshall für das All-Pairs-Shortest-Paths Problem. Dort wurde der Wert einer optimalen Lösung für Problem P als einfache (Minimum-) Funktion der Werte optimaler Lösungen von kleineren (bzw. eingeschränkten) Problemen ausgedrückt. Dieses Prinzip nennt man auch *Bellmansche Optimalitätsgleichung*.

Wir betrachten im Folgenden eine effiziente Dynamische Programmierung für das 0/1-Rucksackproblem.

1.5.1 Dynamische Programmierung für das 0/1-Rucksackproblem

In diesem Abschnitt behandeln wir das 0/1-Rucksackproblem bei dem alle Daten ganzzahlig sind, d.h. die Gewichte w_i und die Werte c_i der $n = N$ Gegenstände sowie die Rucksackgröße K sind ganzzahlig.

Eine Möglichkeit, das 0/1-Rucksackproblem aus Abschnitt 1.3.1 in Teilprobleme zu zerlegen, ist die folgende: Wir betrachten zunächst die Situation nach der Entscheidung über die ersten i Gegenstände. Wir werden sehen, wie man aus den guten Lösungen des eingeschränkten Problems mit i Objekten gute Lösungen des Problems mit $i + 1$ Objekten erhält.

Definition 1.7. Für ein festes $i \in \{1, \ldots, n\}$ und ein festes $W \in \{0, \ldots, K\}$ sei $R(i, W)$ das eingeschränkte Rucksackproblem mit den ersten i Objekten, deren Gewichte $w_1, \ldots, w_i$ und deren Werte $c_1, \ldots, c_i$ betragen und bei dem das Gewichtslimit W beträgt.

Wir legen eine Tabelle $T(i, W)$ an mit $i + 1$ Zeilen für $j = 0, \ldots, i$ und $W + 1$ Spalten für $j = 0, \ldots, W$, wobei an Position $T(i, W)$ folgende Werte gespeichert werden:

- $F(i, W)$: der optimale Lösungswert für Problem $R(i, W)$
- $D(i, W)$: die dazugehörige optimale Entscheidung über das i-te Objekt

Dabei setzen wir $D(i, W) = 1$, wenn es in $R(i, W)$ optimal ist, das i-te Element einzupacken, und $D(i, W) = 0$ sonst. Der Wert einer optimalen Rucksackbepackung für das Originalproblem ist dann gegeben durch $F(n, K)$.

Wir studieren nun, wie für ein gegebenes i und W der Wert $F(i, W)$ aus bereits bekannten Lösungswerten $F(i - 1, W')$ ($W' \leq W$) berechnet werden kann. Hierbei etrachten wir zwei Fälle:

Eingabe: n Gegenstände mit Gewichten $w_1, \ldots, w_n$ und Werten $c_1, \ldots, c_n$; Rucksack der Größe K
Ausgabe: Optimaler Lösungswert mit ihrem Gesamtwert und Gesamtgewicht

```
1:  var 2-dimensionales Array F[i, W] für i = 0, ..., n und W = 0, ..., K
2:  for W = 0, ..., K do
3:      F[0, W] := 0
4:  end for                                              ▷ Initialisierung
5:  for i = 1, ..., n do
6:      for W = 0, ..., w_i − 1 do
7:          F(i, W) := F(i − 1, W)
8:          D(i, W) := 0
9:      end for
10:     for W = w_i, ..., K do
11:         if F(i − 1, W − w_i) + c_i > F(i − 1, W) then
12:             F(i, W) := F(i − 1, W − w_i) + c_i; D(i, W) := 1
13:         else F(i, W) := F(i − 1, W); D(i, W) := 0
14:         end if
15:     end for
16: end for
```

Listing 1.6: Rucksack Dynamische Programmierung

- 1. Fall: Das i-te Element wird eingepackt: In diesem Fall setzt sich die neue Lösung $F(i, W)$ aus der optimalen Lösung von $R(i-1, W-w_i)$ und dem Wert des i-ten Elements zusammen.

- 2. Fall: Das i-te Element wird nicht eingepackt: Dann erhält man den gleichen Lösungswert wie für $R(i-1, W)$, also $F(i-1, W)$.

Damit haben wir wieder eine Bellmansche Optimalitätsgleichung gegeben und können die neuen Lösungswerte effizient berechnen:

$$F(i, W) := \max\{F(i-1, W-w_i) + c_i), F(i-1, W)\}$$

Für die Berechnung setzen wir die Anfangswerte $F(0, W) := 0$ für alle $W = 0, \ldots, K$. Für kleine W passiert es oft, dass $w_i > W$ gilt; in diesem Fall paßt Gegenstand i nicht in den auf W eingeschränkten Rucksack. Listing 1.6 enthält den Pseudocode zur Dynamischen Programmierung für das Rucksackproblem.

Beispiel 1.7. Beispiel: Es sei ein Rucksack der Größe $K = 9$ gegeben und $n = 7$ Gegenstände mit folgenden Gewichten und Werten.

Gegenstand i	1	2	3	4	5	6	7
Wert c_i	6	5	8	9	6	7	3
Gewicht w_i	2	3	6	7	5	9	4

Die Dynamische Programmierungstabelle der Lösungswerte $F(i, W)$ ergibt sich wie folgt.

W/i	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	11	11	11	11	11	12	14	15
6	0	0	11	11	11	11	11	12	14	15
7	0	0	11	11	11	12	12	12	14	15

Die Korrektheit ergibt sich direkt aus der Bellmannschen Optimalitätsgleichung.

Wie können wir nun aus dem optimalen Lösungswert die Lösung selbst, d.h. die einzelnen dazugehörigen Entscheidungen berechnen? In $T[i, W]$ sind die optimalen Lösungswerte Werte $F(i, W)$ gespeichert. An der Stelle $F(n, K)$ steht also der optimale Lösungswert für unser Originalproblem. Wir starten bei $F(n, K)$. Falls $D(n, K) = 0$, dann packen wir Gegenstand n nicht ein. Wir gehen weiter zu Problem $R(n-1, K)$. Falls $D(n, K) = 1$, dann packen wir Gegenstand n ein. Damit ist das für die ersten $n-1$ Gegenstände erlaubte Gewichtslimit $K - w_n$. Gehe weiter zu Problem $R(n-1, K-w_n)$. Wiederhole dies bis $R(0, 0)$ erreicht ist.

Beispiel 1.8. Für unser Beispiel bedeutet dies, dass man von $R(7, 9)$ nach $R(4, 9)$ wandert, wo Gegenstand 4 eingepackt wurde, und von dort über $R(3, 2)$ nach $R(1, 2)$ wandert, wo Gegenstand 1 eingepackt wurde. Danach gelangt man zu $R(0, 0)$ ohne einen weiteren Gegenstand einzupacken. Damit hat man die optimale Lösung (Gegenstände 1 und 4) berechnet.

Analyse der Laufzeit. Die Laufzeit der Berechnung der Werte $F(i, W)$ und $D(i, W)$ ist $O(nK)$ für n Gegenstände und Rucksackgröße K. Die Laufzeit der Berechnung der Lösung ist $O(n)$.

Oberflächlich betrachtet sieht die Laufzeit $O(nK)$ polynomiell aus. Aber VORSICHT, dies ist nicht der Fall, denn die Rechenzeit muss auf die Länge (genauer Bitlänge) der Eingabe

bezogen werden. Wenn alle Zahlen der Eingabe nicht größer sind als 2^n und $K = 2^n$, dann ist die Länge der Eingabe $\theta(n^2)$, denn wir haben $2n+1$ Eingabezahlen mit Kodierungslänge je $\theta(n)$. Die Laufzeit ist jedoch von der Größenordnung $N2^n$ und somit exponentiell in der Inputlänge (also nicht durch ein Polynom beschränkt). Falls aber alle Zahlen der Eingabe in $O(n^2)$ sind. dann liegt die Eingabelänge im Bereich zwischen $\Omega(n)$ und $O(n \log n)$. Dann ist die Laufzeit in $O(n^3)$ und damit polynomiell.

Definition 1.8. Rechenzeiten, die exponentiell sein können, aber bei polynomiell kleinen Zahlen in der Eingabe polynomiell sind, heißen *pseudopolynomiell*.

InsbesondereAlgorithmen mit einer Laufzeit, die direkt von den Eingabewerten und nicht nur von der Anzahl der eingegebenen Objekte abhängen, sind pseudopolynomiell.

Theorem 1.5. *Das 0/1-Rucksackproblem kann mit Hilfe von Dynamischer Programmierung in pseudopolynomieller Rechenzeit gelöst werden.*

In der Praxis kann man Rucksackprobleme meist mit Hilfe von Dynamischer Programmierung effizient lösen, obwohl das Problem NP-schwierig ist, da die auftauchenden Zahlen relativ klein sind.

Das besprochene DP-Verfahren heißt *Dynamische Programmierung durch Gewichte*. Denn wir betrachten die Lösungswerte als Funktion der Restkapazitäten im Rucksack. Durch Vertauschung der Rollen von Gewicht und Wert kann auch *Dynamische Programmierung durch Werte* durchgeführt werden. Dort betrachtet man alle möglichen Lösungswerte und überlegt, wie man diese mit möglichst wenig Gewicht erreichen kann.

1.6 Verbesserungsheuristiken

Wir schließen das Kapitel Optimierung mit allgemeinen, beliebten und weit verbreiteten Heuristiken ab, die versuchen eine gegebene Lösung zu verbessern.

Die in Abschnitt 1.1 gezeigten Heuristiken werden auch als *konstruktive Heuristiken* bezeichnet, da sie neue Lösungen von Grund auf generieren. Im Gegensatz dazu gibt es auch Verfahren, die eine zulässige Lösung als Input verlangen und diese durch lokale Änderungen verbessern. Die Ausgangslösung kann hierbei eine zufällig erzeugte, gültige Lösung sein oder aber durch eine vorangestellte konstruktive Heuristik erzeugt werden.

Wir unterscheiden zwischen einfacher lokaler Suche (s. Abschnitt 1.6.1) und sogenannten Meta-Heuristiken wie Simulated Annealing (s. Abschnitt 1.6.2) und evolutionäre Algorithmen (s. Abschnitt 1.6.3). Aus Zeitgründen betrachten wir alternative Methoden wie z.B. Tabu Suche oder Ameisenverfahren nur sehr kurz (s. Abschnitt 1.6.4).

1.6.1 Einfache lokale Suche

Für die einfache lokale Suche ist die Definition der *Nachbarschaft* $N(x)$ einer Lösung x wesentlich. Das ist die Menge von Lösungen, die von einer aktuellen Lösung x aus durch eine kleine Änderung – einen sogenannten *Zug* – erreichbar sind.

Werden Lösungen beispielsweise durch Bitvektoren repräsentiert, wie das im Rucksackproblem der Fall ist, so könnte $N(x)$ die Menge aller Bitvektoren sein, die sich von x in genau einem Bit unterscheiden. Größere Nachbarschaften können definiert werden, indem man alle Lösungen, die sich in maximal r Bits von x unterscheiden, als Nachbarn betrachtet. Dabei ist r eine vorgegebene Konstante.

Etwas allgemeiner kann eine sinnvolle Nachbarschaft für ein gegebenes Problem oft wie folgt durch *Austausch* definiert werden: Entferne aus der aktuellen Lösung x bis zu r Lösungselemente. Sei S die Menge aller verbleibenden Elemente der Lösung. Die Nachbarschaft von x besteht nun aus allen zulässigen Lösungen, die S beinhalten.

Eingabe: eine Optimierungsaufgabe
Ausgabe: heuristische Lösung x
1: **var** Nachbarlösung x' zu aktueller Lösung x
2: $x =$ Ausgangslösung;
3: **repeat**
4: Wähle $x' \in N(x)$; ▷ leite eine Nachbarlösung ab
5: **if** x' besser als x **then**
6: $x = x'$;
7: **end if**
8: **until** Abbruchkriterium erfüllt

Listing 1.7: Einfache lokale Suche

Algorithmus 1.7 zeigt das Prinzip der einfachen lokalen Suche. In Zeile 3 wird aus der Nachbarschaft eine neue Lösung x' ausgewählt. Diese Auswahl kann auf folgende Art erfolgen.

Random neighbor: Es wird eine Nachbarlösung zufällig abgeleitet. Diese Variante ist die schnellste, jedoch ist die neue Lösung x' häufig schlechter als x.

Best improvement: Die Nachbarschaft $N(x)$ wird vollständig durchsucht und die beste Lösung x' wird ausgewählt. Dieser Ansatz ist am zeitaufwändigsten.

First improvement: Die Nachbarschaft $N(x)$ wird systematisch durchsucht, bis die erste Lösung, welche besser als x ist, gefunden wird bzw. alle Nachbarn betrachtet wurden.

Bei lokaler Suche mit der „random neighbor“ Auswahlstrategie sind normalerweise wesentlich mehr Iterationen notwendig, um eine gute Endlösung zu finden, als bei den beiden

anderen Strategien. Dieser Umstand wird aber oft durch den geringen Aufwand einer einzelnen Iteration ausgeglichen. Welche Strategie in der Praxis am Besten ist, ist vor allem auch vom konkreten Problem abhängig.

In den Zeilen 4 bis 6 von Algorithmus 1.7 wird x' als neue aktuelle Lösung akzeptiert, wenn sie besser ist als die bisherige.

Das Abbruchkriterium ist meist, dass in $N(x)$ keine Lösung mehr existiert, die besser als die aktuelle Lösung x ist. Eine solche Endlösung kann durch Fortsetzung der lokalen Suche nicht mehr weiter verbessert werden und wird daher auch als *lokales Optimum* in Bezug auf $N(x)$ bezeichnet. Formal gilt für eine zu minimierende Zielfunktion $f(x)$:

$$x \text{ ist ein lokales Optimum} \quad \leftrightarrow \quad \forall x' \in N(x) \mid f(x') \geq f(x)$$

Im Allgemeinen ist ein lokales Optimum aber nicht unbedingt ein globales Optimum, an dem wir eigentlich interessiert sind.

Beispiel: Zweier-Austausch für das Symmetrische TSP (2-OPT)

Es wird versucht, eine Ausgangstour iterativ zu verbessern, indem Paare von Kanten temporär entfernt werden und die so resultierenden zwei Pfade durch neue Kanten verbunden werden. Abbildung 1.5 zeigt ein Beispiel. In folgendem Algorithmus wird die Nachbarschaft $N(T)$ einer Tour T systematisch nach einer Verbesserung durchforstet. Die erste Verbesserung wird akzeptiert, was der Auswahlstrategie „first improvement“ entspricht. Im Schritt (2) werden speziell nur die Paare von in der Tour *nicht nebeneinander liegenden* Kanten ausgewählt, da das Entfernen von nebeneinander liegenden Kanten nie zu einer neuen Tour führen kann.

(1) Wähle eine beliebige Anfangstour $T = \{(i_1, i_2), (i_2, i_3), \ldots, (i_n, i_1)\}$; sei $i_{n+1} = i_1$

(2) Setze $Z = \{\{(i_p, i_{p+1}), (i_q, i_{q+1})\} \subset T \mid 1 \leq p, q \leq n \wedge p+1 < q \wedge q+1 \bmod n \neq p\}$

(3) Für alle Kantenpaare $\{(i_p, i_{p+1}), (i_q, i_{q+1})\}$ aus Z:
Falls $c_{i_p i_{p+1}} + c_{i_q i_{q+1}} > c_{i_p i_q} + c_{i_{p+1} i_{q+1}}$:
setze $T = (T \backslash \{(i_p, i_{p+1}), (i_q, i_{q+1})\}) \cup \{(i_p, i_q), (i_{p+1}, i_{q+1})\}$
gehe zu (2)

(4) T ist das Ergebnis.

Beispiel: r-Austausch für das Symmetrische TSP (r-OPT)

Bei dieser Verallgemeinerung werden nicht nur Paare von Kanten durch neue Kanten ersetzt, sondern systematisch alle r-Tupel.

(1) Wähle eine beliebige Anfangstour $T = \{(i_1, i_2), (i_2, i_3), \ldots, (i_n, i_1)\}$

(2) Sei Z die Menge aller r-elementigen Teilmengen von T

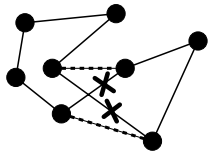


Abbildung 1.5: Prinzip eines Zweieraustausches beim symmetrischen TSP.

(3) Für alle $R \in Z$:
Setze $S = T \setminus R$ und konstruiere alle Touren, die S enthalten. Gibt es eine unter diesen, die besser als T ist, wird diese das aktuelle T und gehe zu (2).

(4) T ist das Ergebnis.

Eine Tour, die durch einen r-Austausch nicht mehr verbessert werden kann, heißt in diesem Zusammenhang r-*optimal*.

Einfache lokale Suche ist eine in der Praxis sehr beliebte Methode, um bereits vorhandene Lösungen oft deutlich und in relativ kurzer Zeit zu verbessern.

Anmerkung 1.5. Für das TSP kommt 3-Opt meist sehr nahe an die optimale Lösung (ca. 3-4%). Jedoch erhält man bereits für $N \geq 200$ Städte sehr hohe Rechenzeiten, da der Aufwand pro Iteration bei $O(N^3)$ liegt. In der Praxis hat sich für das TSP als beste Heuristik die Heuristik von Lin und Kernighan (1973) herausgestellt, die oft bis zu 1-2% nahe an die Optimallösung herankommt. Bei dieser Heuristik werden generierte Kandidatenlösungen analysiert und r wird aus der aktuellen Situation heraus dynamisch gewählt. Ein solcher Ansatz wird allgemein auch *variable Tiefensuche* genannt.

1.6.2 Simulated Annealing

Einfachen lokale Sucheverfahren wie 2-Opt ist es oft nicht möglich, mitunter schlechten lokalen Optima zu „entkommen“. Eine Vergrößerung der Nachbarschaft kann dieses Problem manchmal lösen, es steigt aber der Aufwand meist allzu stark mit der Problemgröße an. *Simulated Annealing* ist eine allgemeine Erweiterung der einfachen lokalen Suche (eine *Meta-Heuristik*), die es bei skalierbarem Zeitaufwand erlaubt, lokalen Optima grundsätzlich auch zu entkommen, ohne dass die Nachbarschaft vergrößert werden muss.

Abgeleitet wurde dieses Verfahren von dem physikalischen Prozess, ein Metall in einen Zustand möglichst geringer innerer Energie zu bringen, in dem die Teilchen möglichst strukturiert angeordnet sind. Dabei wird das Metall erhitzt und sehr langsam abgekühlt. Die Teilchen verändern anfangs ihre Positionen und damit ihre Energieniveaus sehr stark, mit sinkender Temperatur werden die Bewegungen von lokal niedrigen Energieniveaus weg aber immer geringer.

Eingabe: eine Optimierungsaufgabe
Ausgabe: heuristische Lösung x

```
1:  var Zeit t; aktuelle Temperatur T; Ausgangstemperatur T_init; Nachbarlösung x'
2:  t = 0;
3:  T = T_init;
4:  x = Ausgangslösung;
5:  repeat
6:      Wähle x' ∈ N(x) zufällig;                    ▷ leite eine Nachbarlösung ab
7:      if x' besser als x then
8:          x = x';
9:      else
10:         if Z < e^{-|f(x')-f(x)|/T} then
11:             x = x';
12:         end if
13:         T = g(T, t);
14:         t = t + 1;
15:     end if
16: until Abbruchkriterium erfüllt
```

Listing 1.8: Simulated Annealing

Algorithmus 1.8 zeigt das von Kirkpatrick et al. (1983) vorgeschlagene Simulated Annealing. Z ist hierbei eine Zufallszahl $\in [0, 1)$, $f(x) > 0$ die Bewertung der Lösung x.

Die Ausgangslösung kann wiederum entweder zufällig oder mit einer Konstruktionsheuristik generiert werden. In jeder Iteration wird dann ausgehend von der aktuellen Lösung x eine Nachbarlösung x' durch eine kleine zufällige Änderung abgeleitet, was der Auswahlstrategie „random neighbor“ entspricht. Ist x' besser als x, so wird x' in jedem Fall als neue aktuelle Lösung akzeptiert. Ansonsten, wenn also x' eine schlechtere Lösung darstellt, wird diese mit einer Wahrscheinlichkeit $p_{accept} = e^{-|f(x')-f(x)|/T}$ akzeptiert. T ist dabei die aktuelle „Temperatur“. Dadurch ist die Möglichkeit gegeben, lokalen Optima zu entkommen. Diese Art, Nachbarlösungen zu akzeptieren wird in der Literatur auch *Metropolis-Kriterium* genannt.

Die Wahrscheinlichkeit p_{accept} hängt von der Differenz der Bewertungen von x und x' ab – nur geringfügig schlechtere Lösungen werden mit höherer Wahrscheinlichkeit akzeptiert als viel schlechtere. Außerdem spielt die Temperatur T eine Rolle, die über die Zeit hinweg kleiner wird: Anfangs werden Änderungen mit größerer Wahrscheinlichkeit erlaubt als später.

Wie T initialisiert und in Abhängigkeit der Zeit t vermindert wird, beschreibt das *Cooling-Schema*.

Geometrisches Cooling: Für T_{init} wird z.B. $f_{\max} - f_{\min}$ gewählt. Sind $f_{\max}$ bzw. $f_{\min}$ nicht bekannt, so werden Schranken bzw. Schätzungen hierfür verwendet. Als Cooling-Funktion wird z.B. gewählt $g(T, t) = T \cdot \alpha, \quad \alpha < 1$ (z.B. 0,999).

Adaptives Cooling: Es wird der Anteil der Verbesserungen an den letzten erzeugten Lösungen gemessen und auf Grund dessen T stärker oder schwächer reduziert.

Als Abbruchkriterium sind unterschiedliche Bedingungen vorstellbar:
Ablauf einer bestimmten Zeit; eine gefundene Lösung ist „gut genug“; oder keine Verbesserung in den letzten k Iterationen.

Damit Simulated Annealing grundsätzlich funktioniert und Chancen bestehen, das globale Optimum zu finden, müssen folgende zwei Bedingungen erfüllt sein.

- **Erreichbarkeit eines Optimums**: Ausgehend von jeder möglichen Lösung muss eine optimale Lösung durch wiederholte Anwendung der Nachbarschaftsfunktion grundsätzlich mit einer Wahrscheinlichkeit größer Null erreichbar sein.
- **Lokalitätsbedingung:** Eine Nachbarlösung muss aus ihrer Ausgangslösung durch eine „kleine“ Änderung abgeleitet werden können. Dieser kleine Unterschied muss im Allgemeinen (d.h. mit Ausnahmen) auch eine kleine Änderung der Bewertung bewirken. Sind diese Voraussetzungen erfüllt, spricht man von *hoher Lokalität* – eine effiziente Optimierung ist möglich. Haben eine Ausgangslösung und ihre abgeleitete Nachbarlösung im Allgemeinen große Unterschiede in der Bewertung, ist die Lokalität schwach. Die Optimierung ähnelt dann der Suche „einer Nadel im Heuhaufen“ bzw. einer reinen Zufallssuche und ist nicht effizient.

Wir betrachten ein Beispiel für sinnvolle Operatoren, um eine Nachbarlösung abzuleiten.

Beispiel: Symmetrisches TSP. *Operator Inversion:* Ähnlich wie im 2-Opt werden zwei nicht benachbarte Kanten einer aktuellen Tour zufällig ausgewählt und entfernt. Die so verbleibenden zwei Pfade werden mit zwei neuen Kanten zu einer neuen Tour zusammengefügt.

Anmerkung 1.6. Dass dieser Operator keinerlei Problemwissen wie etwa Kantenkosten ausnutzt, ist einerseits eine Schwäche; andererseits ist das Verfahren aber so auch für schwierigere Varianten des TSPs, wie dem *blinden TSP*, einsetzbar. Bei dieser Variante sind die Kosten einer Tour nicht einfach die Summe fixer Kantenkosten, sondern im Allgemeinen eine nicht näher bekannte oder komplizierte, oft nicht-lineare Funktion. Beispielsweise können die Kosten einer Verbindung davon abhängen, *wann* diese verwendet wird. (Wenn Sie mit einem PKW einerseits in der Stoßzeit, andererseits bei Nacht durch die Stadt fahren, wissen Sie, was gemeint ist.)

1.6.3 Evolutionäre Algorithmen

Unter dem Begriff *evolutionäre Algorithmen* werden eine Reihe von Meta-Heuristiken (genetische Algorithmen, Evolutionsstrategien, Evolutionary Programming, Genetic Programming, etc.) zusammengefasst, die Grundprinzipien der natürlichen Evolution in einfacher

Weise nachahmen. Konkret sind diese Mechanismen vor allem die *Selektion* (natürliche Auslese, „survival of the fittest“), die *Rekombination* (Kreuzung) und die *Mutation* (kleine, zufällige Änderungen).

Ein wesentlicher Unterschied zu Simulated Annealing ist, dass nun nicht mehr mit nur einer aktuellen Lösung, sondern einer ganze Menge (= *Population*) gearbeitet wird. Durch diese größere *Vielfalt* ist die Suche nach einer optimalen Lösung „breiter“ und robuster, d.h. die Chance lokalen Optima zu entkommen ist größer.

Algorithmus 1.9 zeigt das Schema eines evolutionären Algorithmus.

Eingabe: eine Optimierungsaufgabe
Ausgabe: beste gefundene Lösung
1: **var** selektierte Eltern Q_s; Zwischenlösungen Q_r
2: P = Menge von Ausgangslösungen;
3: bewerte(P);
4: **repeat**
5: Q_s = Selektion(P);
6: Q_r = Rekombination(Q_s);
7: P = Mutation(Q_r);
8: bewerte(P);
9: **until** Abbruchkriterium erfüllt

Listing 1.9: Grundprinzip eines evolutionären Algorithmus

Ausgangslösungen können wiederum entweder zufällig oder mit einfachen Erzeugungsheuristiken generiert werden. Wichtig ist jedoch, dass sich diese Lösungen unterscheiden und so Vielfalt gegeben ist.

Selektion. Die Selektion kopiert aus der aktuellen Population P Lösungen, die in den weiteren Schritten neue Nachkommen produzieren werden. Dabei werden grundsätzlich bessere Lösungen mit höherer Wahrscheinlichkeit gewählt. Schlechtere Lösungen haben aber im Allgemeinen auch Chancen, selektiert zu werden, damit lokalen Optima entkommen werden kann.

Eine häufig eingesetzte Variante ist die **Tournament Selektion**, die wie folgt funktioniert:

(1) Wähle aus der Population k Lösungen gleichverteilt zufällig aus (Mehrfachauswahl ist üblicherweise erlaubt).

(2) Die beste der k Lösungen ist die selektierte.

Rekombination. Die Aufgabe der Rekombination ist, aus zwei selektierten Elternlösungen eine neue Lösung zu generieren. Vergleichbar mit der bereits beim Simulated Annealing

beschriebenen Lokalitätsbedingung ist hier wichtig, dass die neue Lösung möglichst ausschließlich aus Merkmalen der Eltern aufgebaut wird.

Mutation. Die Mutation entspricht der Nachbarschaftsfunktion beim Simulated-Annealing. Sie dient meist dazu, neue oder verlorengegangene Merkmale in die Population hineinzubringen.

Häufig werden die Rekombination und Mutation nicht immer, sondern nur mit einer bestimmten Wahrscheinlichkeit ausgeführt, sodass vor allem gute Lösungen manchmal auch unverändert in die Nachfolgegeneration übernommen werden.

Wir sehen uns ein konkretes Beispiel an.

Beispiel: Symmetrisches TSP. *Edge-Recombination* (ERX): Aus den Kanten zweier Elterntouren T^1 und T^2 soll eine neue Tour T erstellt werden, die möglichst nur aus Kanten der Eltern aufgebaut ist. Algorithmus 1.10 zeigt ein mögliches Vorgehen.

Eingabe: Zwei gültige Touren T^1 und T^2
Ausgabe: Neue abgeleitete Tour T
1: **var** aktueller Knoten v; Nachfolgeknoten w; Kandidatenmenge für Nachfolgeknoten W
2: beginne bei einem beliebigen Startknoten $v = v_0$; $T = \{\}$;
3: **while** es noch unbesuchte Knoten gibt **do**
4: Sei W die Menge der noch unbesuchten Knoten, die in $T^1 \cup T^2$ adjazent zu v sind;
5: **if** $W \neq \{\}$ **then**
6: wähle einen Nachfolgeknoten $w \in W$ zufällig aus;
7: **else**
8: wähle einen zufälligen noch nicht besuchten Nachfolgeknoten w;
9: **end if**
10: $T = T \cup \{(v, w)\}$; $v = w$;
11: **end while**
12: schließe die Tour: $T = T \cup \{(v, v_0)\}$;

Listing 1.10: Edge-Recombination(T^1, T^2)

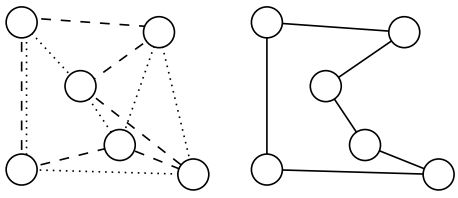


Abbildung 1.6: Beispiel zur Edge-Recombination.

Abb. 1.6 zeigt links zwei übereinandergelegte Elterntouren und rechts eine mögliche, durch Edge-Recombination neu erzeugte Tour, die nur aus Kanten der Eltern besteht.

Im Algorithmus werden neue Kanten, die nicht von den Eltern stammen, nur dann zu T hinzugefügt, wenn W leer ist („edge-fault“). Um die Wahrscheinlichkeit, dass es zu diesen Situationen kommt, möglichst gering zu halten, kann die Auswahl in Schritt (5) wie folgt verbessert werden: Ermittle für jeden Knoten $w_i \in W$ die Anzahl der gültigen Knoten, die von diesem dann weiter unter Verwendung von Elternkanten angelaufen werden können, d.h. $a_i = |\{u \mid (\{w_i, u\} \in T^1 \cup T^2 \wedge u \text{ noch nicht besucht in } T\}|$.
Wähle ein $w = w_i$ für das a_i minimal ist.

In einer anderen Variante der Edge-Recombination werden die Kantenlängen als lokale, problem-spezifische Heuristik mitberücksichtigt: In Schritt 5 wird entweder immer oder mit höherer Wahrscheinlichkeit die Kante kürzester Länge ausgewählt. Dies führt einerseits zu einer rascheren Konvergenz des evolutionären Algorithmus zu guten Lösungen, kann aber auch ein vorzeitiges „Hängenbleiben“ bei weniger guten, lokal-optimalen Touren bewirken.

Anmerkung 1.7. Abschließend sei zu evolutionären Algorithmen noch angemerkt, dass sie mit anderen Heuristiken und lokalen Verbesserungstechniken sehr effektiv kombiniert werden können. So ist es möglich

- Ausgangslösungen mit anderen Heuristiken zu erzeugen,
- in der Rekombination und Mutation Heuristiken einzusetzen (z.B. können beim TSP kostengünstige Kanten mit höherer Wahrscheinlichkeit verwendet werden), und
- alle (oder einige) erzeugte Lösungen mit einem anderen Verfahren (z.B. 2-Opt) noch lokal weiter zu verbessern.

Auch können die Vorteile paralleler Hardware gut genutzt werden.

1.6.4 Alternative Heuristiken

Es gibt noch einige andere Klassen von sogenannten Meta-Heuristiken:

- Eine beliebte und in vielen Fällen auch sehr erfolgreiche Alternative zu Simulated Annealing stellt die 1986 von Glover vorgestellte *Tabu-Suche* dar. Auch sie ist eine Erweiterung der einfachen lokalen Suche, die darauf abzielt, lokalen Optima grundsätzlich entkommen zu können, um ein globales Optimum zu finden. Die Tabu-Suche verwendet ein Gedächtnis über den bisherigen Optimierungsverlauf und nutzt dieses, um bereits erforschte Gebiete des Suchraums nicht gleich nochmals zu betrachten.
- Ameisen-Verfahren (*Ant Colony, ACO*) wurden ursprünglich für Wegeprobleme (z.B. TSP) entwickelt und basieren auf der Idee der Pheromon-Spuren, die Ameisen auf ihren Wegen hinterlassen. Die Idee dabei ist, dass Ameisen, die einen kurzen Weg zur Futterquelle nehmen, diesen öfter hin-und-her laufen, und dort also stärkere Pheromonspuren hinterlassen, wovon andere Ameisen angezogen werden. Allerdings sind ACO Verfahren meist sehr langsam.

- Sintflut-Verfahren entsprichtt der Idee von Simulated Annealing, allerdings sorgt hier der steigende Wasserpegel (bei SA entspricht dies der Temperatur) dafür, dass schlechtere Nachbarlösungen mit zunehmender Laufzeit des Verfahrens seltener akzeptiert werden. D.h. i.W. ist der Zufall Z von Simulated Annealing hier ausgeschaltet.

- Für Schlagzeilen sorgte vor einiger Zeit der Artikel über DNA-Computing, bei dem ein kleines TSP "im Reagenzglas" mit linear vielen Schritten gelöst wurde. Inzwischen ist die Euphorie abgeklungen, denn statt exponentiell viel Zeit benötigt dieser Ansatz exponentiell viel Platz. Um z.B ein 100-Städte TSP damit zu lösen, müßte ein nichtrealisierbares riesengroßes Reagenzglas gebaut werden...

Weiterführende Literatur

- C.H. Papadimitriou und K. Steiglitz: „Combinatorial Optimization: Algorithms and Complexity“, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982

- W.J. Cook, W.H. Cunningham, W.R. Pulleyblank und A. Schrijver: „Combinatorial Optimization“, John Wiley & Sons, Chichester, 1998

- E. Aarts und J.K. Lenstra: „Local Search in Combinatorial Optimization“, John Wiley & Sons, Chichester, 1997

- Z. Michalewicz: „Genetic Algorithms + Data Structures = Evolution Programs“, Springer, 1996