

Optimization

R. Koller, G. Danuser, S.Eicher

1. Juli 2024

Inhaltsverzeichnis

1	Mathematische Modelle	Skript S. 27	3
1.1	Deskriptive Modelle und Optimierungsmodelle	Skript S. 27	3
1.1.1	Beispiel für Modellierung: Warenverteilung (Ähnlich, aber nicht identisch zu Klinkerts Mietwagenvermietung, Übung 1, Aufgabe 2 bzw. Skript S. 23)		3
1.1.2	Beispiel für Modellierung: Legierungen (Klinkert, Übung 4, Aufgabe 5)		3
1.2	Allgemeines Optimierungsproblem	Skript S. 31	4
1.3	Konvexe Optimierung	Skript S. 40	5
1.4	Mengenlehre		5
1.5	Typen von Optimierungsmodellen und -methoden	Skript S. 46	5
2	Lineare Programmierung	Skript S. 49	7
2.1	Problemformulierung		7
2.1.1	Notationen		7
2.1.2	Normalformen		7
2.1.3	Umformulierungen	Skript S. 51	8
2.1.4	Rang einer Matrix / Eckpunkte einer Matrix		8
2.1.5	Graphische Darstellung		8
2.1.6	Hyperebenen		8
2.1.7	Basisauswahl		8
2.1.8	Basislösung		9
3	Simplex-Algorithmus		9
3.1	Beispiel:		9
3.1.1	Algorithmus		10
4	Ganzzahlig-lineare Programmierung	Skript S. 63	11
4.1	Relaxationen		11
4.2	Branch-and-Bound Verfahren	Skript S. 68	11
4.2.1	Beispiel: ILP Übung 6.2		11
4.2.2	Beispiel: Branch-and-bound beim Knapsack-Problem	Skript S. 73–78	11
4.3	Schnittebenenverfahren (Cutting Planes)	Skript S. 78	12
4.3.1	Chvátal-Rang berechnen	Skript S. 91	12
5	Nichtlineare Optimierung		13
5.1	Gradient	Skript S. 3	13
5.2	Algebraisch Extrema bestimmen		13
5.3	Gradientenverfahren	Skript S. 4	13
5.3.1	Eigenschaften		13
5.3.2	Algorithmus (zur Minimierung)		13
5.4	Differential-Rechnung		13
5.4.1	Matlab-Implementation (nur ein Loop)		13
5.5	Newton-Verfahren	Skript S. 7	14
5.6	Quasi-Newton-Verfahren	Skript S. 11	15
5.7	Konvergenzbeschleunigung	Skript S. 17	15

6 Graphen und Netzwerke	16
6.1 Repräsentation von Graphen Skript S. 3	16
6.2 Entscheidungsbaum-Verfahren Skript S. 4	16
6.3 Traversieren von Graphen Skript S. 9	17
6.3.1 Tiefensuche (Depth-First Search, DFS)	17
6.3.2 Breitensuche (Breadth-First Search, BFS)	17
6.4 Minimum Spanning Tree (MST)	17
6.4.1 Prim-Algorithmus	17
6.4.2 Kruskal-Algorithmus	17
6.4.3 MST-Heuristik zur Lösung des TSP	17
6.5 Minimaler Weg	18
6.5.1 Probleme	18
6.5.2 Algorithmen	18
6.5.3 Dijkstra	18
6.5.4 A*	19
6.5.5 Floyd-Warshall	19
7 Netzwerke	20
7.1 Reduktion auf Maxflow-Problem	20
7.2 Maxflow Problem	21
7.2.1 Augmenting-Path Methode	21
7.2.2 Ford-Fulkerson	21
7.2.3 Max-Flow Min-Cut	21
7.3 Maxflow Mincost Skript S. 28	22
8 Meta-Heuristiken	22
8.1 Hill-Climbing Skript S. 2	22
8.2 Tabu-Search Skript S. 3	22
8.3 Simulated Annealing Skript S. 5	23
8.4 Genetische Algorithmen (GA) Skript S. 8	23
8.4.1 Codierung der Parameter Skript S. 11	24
8.4.2 Selektion Skript S. 15	24
8.4.3 Rekombination Skript S. 16	24
8.4.4 Mutation Skript S. 18	24
8.4.5 Ersetzungsschemata Skript S. 19	25
8.5 Ant Colony Optimization Skript S. 21	25
8.6 Particle Swarm Optimization Skript S. 23	27
9 Optimization Summary	29

1 Mathematische Modelle Skript S. 27

1.1 Deskriptive Modelle und Optimierungsmodelle Skript S. 27

	Deskriptive Modelle	Optimierungsmodelle
Antworten auf...	“What if?”	“What’s best?”
Mengen (I)	Typen von Objekten (z.B. Städte $I = \{1, 2, \dots, n\}$)	
Parameter (p_i)	Vorgegebene Größen des Modells (z.B. aktueller Fahrzeugbestand in Filiale i , Distanz zwischen Filiale i und j)	
Variablen (x_{ij})	Veränderbare Modellgröße (z.B. Anz. transferierte Fahrzeuge von Filiale i nach j)	
Konsequenzen (k_i, k_i^{out})	Output des deskriptiven Modells (z.B. gesamte Fahrdistanz), $k_0 = f_0(x, p)$	-
Zielfunktion min oder max	-	Zu maximierende oder minimierende Konsequenz
Restriktionen / Constraints (k_i, k_i^{out})	-	Bedingungen auf den Konsequenzen (Ungleichungen, selten Gleichungen) ACHTUNG: 'Idiotische' Bedingungen nicht vergessen: Keine negativen Mengen produzieren, etc.

1.1.1 Beispiel für Modellierung: Warenverteilung (Ähnlich, aber nicht identisch zu Klinkerts Mietwagenvermietung, Übung 1, Aufgabe 2 bzw. Skript S. 23)

Eine Ware soll von Lagern an Filialen ausgeliefert werden mit minimalen Transportkosten.

Mengen:

$$\begin{aligned} I & \text{ Menge der Lager, } I = \{1, \dots, m\} \\ J & \text{ Menge der Filialen, } J = \{1, \dots, n\} \end{aligned}$$

Parameter:

$$\begin{aligned} a_i & \text{ Lagerbestand der Ware in Lager } i & i \in I \\ b_j & \text{ Bedarf der Filiale } j & j \in J \\ c_{ij} & \text{ Kosten für Beförderung von Lager } i \text{ nach Filiale } j & i \in I, j \in J \end{aligned}$$

Variablen:

$$x_{ij} \quad \text{Verschobene Menge von } i \text{ nach } j \quad i \in I, j \in J$$

Zielfunktion:

$$\min \sum_{i \in I, j \in J} c_{ij} x_{ij}$$

Restriktionen:

$$\begin{aligned} \sum_{j \in J} x_{ij} & \leq a_i \quad i \in I & (\text{Nicht mehr Waren ausliefern als im Lager vorhanden}) \\ \sum_{i \in I} x_{ij} & \geq b_j \quad j \in J & (\text{Nicht weniger Waren ausliefern als benötigt}) \\ x_{ij} & \geq 0 \quad i \in I, j \in J & (\text{Nicht negative Anzahl Waren ausliefern}) \end{aligned}$$

1.1.2 Beispiel für Modellierung: Legierungen (Klinkert, Übung 4, Aufgabe 5)

Aus den Edelmetallen I sollen die Legierungen J gemischt werden. Der Erlös soll maximiert werden.

Mengen:

$$\begin{aligned} I & \text{ Menge der Edelmetalle, } I = \{1, \dots, m\} \\ J & \text{ Menge der Legierungen, } J = \{1, \dots, n\} \end{aligned}$$

Parameter:

$$\begin{aligned} a_{ij} & \text{ Menge von Edelmetall } i \text{ pro Menge Legierung } j & i \in I, j \in J \\ b_i & \text{ Verfügbare Menge von Edelmetall } i & i \in I \\ c_j & \text{ Erlös pro Menge Legierung } j & j \in J \end{aligned}$$

Variablen:

$$x_j \quad \text{Produktionsmenge von Legierung } j \quad j \in J$$

Zielfunktion:

$$\max \sum_{j \in J} c_j x_j$$

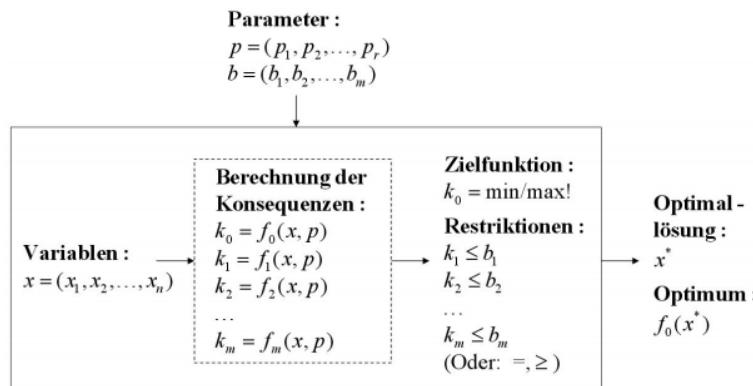
Restriktionen:

$$\sum_{j \in J} a_{ij} x_j \leq b_i \quad i \in I \quad (\text{nicht mehr Legierung produzieren als Edelmetall vorhanden})$$

$$x_j \geq 0 \quad j \in J \quad (\text{nicht negative Mengen Legierungen produzieren})$$

1.2 Allgemeines Optimierungsproblem Skript S. 31

In Form von Ungleichungen und Gleichungen werden mögliche Lösungsmengen (*solution spaces*) definiert. Dort soll eine Zielfunktion f optimiert (maximiert oder minimiert) werden. Die Fragestellung lautet: „What's best?“ (preskriptive Modelle).

**Problemformulierung**

Allgemeines Optimierungsproblem

$$\Pi : \max\{f(x) : x \in S\}$$
 wobei $S \subseteq \mathbb{R}^n$

Entscheidungsvariablen/Variablen (*decision variables/variables*)

$$\text{Vektor } x = (x_1, \dots, x_n) \in \mathbb{R}^n$$

Lösungsmenge/Lösungsraum (*solution space, feasible region*)

$$S \text{ von } \Pi$$

Zielfunktion

$$f : S \rightarrow \mathbb{R}$$

Zulässigkeit (*feasibility*)Lösungsraum darf nicht leer sein ($S \neq \emptyset$)Beschränktheit (*boundedness*)

$$f(x) \leq \omega, \omega \in \mathbb{R} \text{ für alle } x \in S$$

Abgeschlossenheit (*closedness*)Zusätzlich zu Zulässigkeit und Beschränktheit: $x^* \in S$ (Optimallösung existiert)Kontinuität (*continuity*) f ist eine kontinuierliche Funktion**Definition des Lösungsraums**Funktionale Restriktionen (*functional constraints*)

$$p \geq 0 \text{ Ungleichungen der Form } g_i(x) \leq 0,$$

$$q \geq 0 \text{ Ungleichungen der Form } h_j(x) = 0.$$

 $g_i(x), h_j(x)$ sind lineare Funktionen!

Nicht-funktionale Restriktionen

Definition des Lösungsraums S durch „nicht-funktionale“ Restriktionen, z.B. logische Prädikate (z.B. $S = \{x \in \mathbb{Z}^n : x \text{ ist eine Permutation der Zahlen } 1, \dots, n\}$)**Maximierungs-/Minimierungsprobleme**

Extremwert-Theorem von Weierstrasse

Wenn $S \subseteq \mathbb{R}^n$ eine nichtleere, begrenzte, abgeschlossene Menge und $f : S \rightarrow \mathbb{R}$ eine stetige Funktion ist, existiert ein $x^* \in S$ mit $f(x^*) \geq f(x)$ für alle $x \in S$, d.h. das Optimierungsproblem hat eine Optimallösung.**Nachbarschaft (Neighbourhood)**Beliebige Menge Punkte „in der Nähe“ des aktuellen Punktes x

Euklidische Nachbarschaft

$$N_\varepsilon(x) = \{y \in \mathbb{R}^n : \|y - x\| \leq \varepsilon\} \text{ mit } \varepsilon > 0, \|x\| = \sqrt{x^2} = \sqrt{x_1^2 + \dots + x_n^2}$$

(in \mathbb{R}^2 ein Kreis, \mathbb{R}^3 eine Kugel, etc.)

Andere Nachbarschaften

Es sind auch andere Nachbarschaftsdefinitionen möglich (ohne Bsp.)

Diverses

Globale Optimallösung

 x^* ist eine globale Optimallösung wenn $f(x^*) \geq f(x)$ für alle $x \in S$.

Lokale Optima

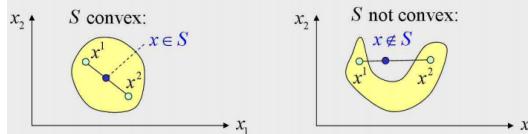
 x^* ist eine lokale Optimallösung (bzgl. Nachbarschaft N_ε) wenn $f(x^*) \geq f(x)$ für alle $x \in N_\varepsilon(x^*) \cap S$ Niveaumengen (Höhenlinien, *level sets*)

Orthogonale Linien zu den linearen Funktionen (Geraden)

1.3 Konvexe Optimierung Skript S. 40

Definitionen

λ	$\lambda \in \mathbb{R}$ mit $0 \leq \lambda \leq 1$
Konvexkombination zweier Vektoren	$x^1, x^2 \in \mathbb{R}^n: x = \lambda x^1 + (1 - \lambda)x^2$
Konvexe Funktion	$f(\lambda x^1 + (1 - \lambda)x^2) \leq \lambda f(x^1) + (1 - \lambda)f(x^2)$ für alle $x^1, x^2 \in S$, S konvex Deutsch: Lineare Interpolation zwischen x^1 und x^2 ist kleiner gleich als Funktion
Konkave Funktion	$f(\lambda x^1 + (1 - \lambda)x^2) \geq \lambda f(x^1) + (1 - \lambda)f(x^2)$ für alle $x^1, x^2 \in S$, S konvex Deutsch: Lineare Interpolation zwischen x^1 und x^2 ist grösser gleich als Funktion
Lineare Funktion	Eine lineare Funktion ist sowohl konvex als auch konkav in S .
Konvexe Menge	$\lambda x^1 + (1 - \lambda)x^2 \in S$ für Vektoren $x^1, x^2 \in S \subseteq \mathbb{R}^n, \lambda \in \mathbb{R}, 0 \leq \lambda \leq 1$.

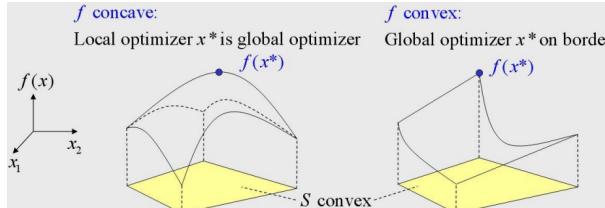


Konvexes Optimierungsproblem

Ein konvexes Optimierungsproblem ist *entweder* die Maximierung einer konkaven Funktion f über einer konvexen Menge S *oder* die Minimierung einer konvexen Funktion f über einer konvexen Menge S .

Eigenschaften

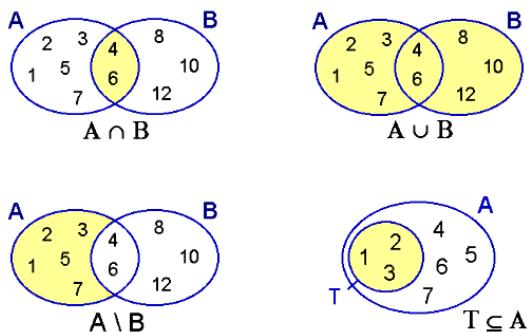
Globales Optimum	In einem konvexen Optimierungsproblem ist jedes lokale Optimum ein globales Optimum!
Optimum auf Rand	$S \subseteq \mathbb{R}^n$ sei eine konvexe, abgeschlossene, begrenzte Menge. Bei einem Optimierungsproblem der Form $\max\{f(x) : x \in S\}$ (bzw. $\min\{f(x) : x \in S\}$) mit f konvex (bzw. konkav) in S liegt jede lokale Optimallösung auf dem Rand.



Lineare Funktion

Für den Fall einer linearen Funktion f über einer konvexen, begrenzten, abgeschlossenen Menge S gilt, dass jede lokale Optimallösung global optimal ist und sich am Rand von S befindet!

1.4 Mengenlehre



1.5 Typen von Optimierungsmodellen und -methoden Skript S. 46

Optimierungsmodelle:

- Constrained vs. Unconstrained
- Global vs. Lokal
- Differenzierbar vs. Nicht-Differenzierbar
- Diskret vs. Kontinuierlich
- Konvex vs. Nicht-Konvex

- Linear vs. Nichtlinear

Optimierungsmethoden:

- Exakt vs. Heuristisch
- Generell vs. Problemspezifisch

Typen von Heuristiken:

- Konstruktiv
- Verbessernde Heuristiken
 - Lokale Suche (*local search*)
 - generelle Meta-Heuristiken (*general meta heuristics*)

2 Lineare Programmierung Skript S. 49

2.1 Problemformulierung

2.1.1 Notationen

Variablen (Spalten)	$x \in \mathbb{R}^n; I = \{1, \dots, m\}$
Constraints ((Un-)gleichungen; Zeilen)	$a^k; J = \{1, \dots, n\}$
Matrix	$A \in \mathbb{R}^{m \times n}$

Lineares Problem (Beispiel U4-1)	$\max x_1 + 3x_2 + 2x_3$	$x_2 + x_3 \leq 2$ $x_1 - 2x_2 \leq -2$	$x_1, x_2, x_3 \geq 0$
Zeilennotation Beispiel	$\max cx$ $c = \text{Zeilenvektor } 1 \times n$ $x = \text{Spaltenvektor } n \times 1$ $\max[1, 3, 2] \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$	$a^i x \leq b_i, i = 1, \dots, m$ $a^i = \text{Zeilenvektor } 1 \times n$ $x = \text{Spaltenvektor } n \times 1$ $a^1 : [0, 1, 1] \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \leq 2$ $a^2 : [1, -2, 0] \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \leq -2$	$x \geq 0$
Spaltennotation Beispiel	$\max cx$ $c = \text{Zeilenvektor } 1 \times n$ $x = \text{Spaltenvektor } n \times 1$ $\max[1, 3, 2] \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$	$\sum_{j=1}^n A_j x_j \leq b$ $A_j = \text{Spaltenvektor } n \times 1$ $b = \text{Spaltenvektor } n \times 1$ $\begin{bmatrix} 0 \\ 1 \end{bmatrix} x_1 + \begin{bmatrix} 1 \\ -2 \end{bmatrix} x_2 + \begin{bmatrix} 1 \\ 0 \end{bmatrix} x_3 \leq \begin{bmatrix} 2 \\ -2 \end{bmatrix}$	$x \geq 0$
Matrixnotation Beispiel	$\max cx$ $c = \text{Zeilenvektor } 1 \times n$ $x = \text{Spaltenvektor } n \times 1$ $\max[1, 3, 2] \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$	$Ax \leq b$ $A = \text{Matrix } m \times n$ $b = \text{Spaltenvektor } m \times 1$ $\begin{bmatrix} 0 & 1 & 1 \\ 1 & -2 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \leq \begin{bmatrix} 2 \\ -2 \end{bmatrix}$	$x \geq 0$

2.1.2 Normalformen

Ein lineares Programm kann in verschiedenen Formen geschrieben werden:

Allgemeine Form	$\max / \min cx$ $a^i x \leq b_i,$ $a^i x = b_i,$ $a^i x \geq b_i,$ $x_j \geq 0,$ $x_j \text{ frei},$ $x_j \leq 0$
Kanonische Form	$\max cx$ $a^i x \leq b_i,$ $x_j \geq 0$ <i>oder</i> $\min cx$ $a^i x \geq b_i,$ $x_j \geq 0$
Standardform	$\max / \min cx$ $a^i x = b_i,$ $x_j \geq 0$
Ungleichungsform	$\max cx$ $a^i x \leq b_i,$ <i>oder</i> $\min cx$ $a^i x \geq b_i,$

Zusätzlich wird zwischen Maximierungs- und Minimierungsproblemen unterschieden.

2.1.3 Umformulierungen Skript S. 51

Achtung: Freie Variable ($x_j = \text{frei}$) und nicht positive ($x_i \leq 0$) als allererstes in allen Formulierungen (Funktion, Bedingungen) ersetzen!

Non-positive to non-negative variables:	$x_j \leq 0 \rightsquigarrow x_j := -\bar{x}_j, \quad \bar{x}_j \geq 0$
Free to non-negative variables:	$x_j \text{ frei} \rightsquigarrow x_j := x_j^+ - x_j^-, \quad x_j^+, x_j^- \geq 0$
Equations to inequalities:	$a^i x = b_i \rightsquigarrow a^i x \leq b_i, \quad a^i x \geq b_i$
Inequalities "\leq" to inequalities "\geq" (and vice versa):	$a^i x \leq b_i \rightsquigarrow -a^i x \geq -b_i \text{ bzw. } a^i x \geq b_i \rightsquigarrow -a^i x \leq -b_i$
Inequalities to equations:	$a^i x \leq b_i \rightsquigarrow a^i x + x_i^s = b_i, \quad x_i^s \geq 0 \text{ bzw. } a^i x \geq b_i \rightsquigarrow a^i x - x_i^s = b_i, \quad x_i^s \geq 0$
$\min\{cx\}$	$\rightsquigarrow \max\{-cx\}$
$\min\{cx : x \in P\}$	$= -\max\{-cx : x \in P\}$
$\max\{cx\}$	$\rightsquigarrow \min\{cx\}$
$\max\{cx : x \in P\}$	$= -\min\{-cx : x \in P\}$ (für Funktionsevaluation)

Beispiel Umformung von Ungleichung in Gleichung: $x_1 + 2x_2 + 4x_3 \geq 12 \rightsquigarrow x_1 + 2x_2 + 4x_3 - x_1^s = 12, \quad x_1^s \geq 0$
 Beispiel Umformung von Gleichung in Ungleichung: $x_1 - x_2 + x_3 = 2 \rightsquigarrow x_1 - x_2 + x_3 \leq 2 \wedge x_1 - x_2 + x_3 \geq 2$

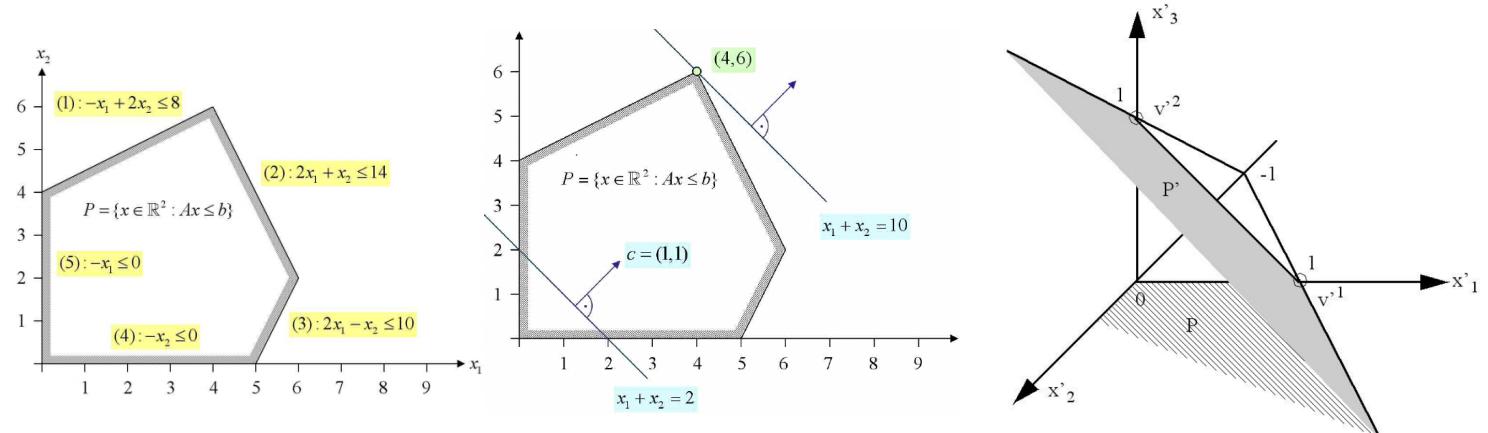
2.1.4 Rang einer Matrix / Eckpunkte einer Matrix

$\text{rank}(A) = \text{Maximal Anzahl linear unabhängiger Spalten. Eine Matrix hat Eckpunkte, wenn sie vollen Spaltenrang hat. Dies}$

ist erfüllt, wenn das Gleichungssystem $\lambda_1 \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \end{bmatrix} + \lambda_2 \begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \end{bmatrix} \dots = \underline{0} = \begin{bmatrix} 0 \\ 0 \\ \vdots \end{bmatrix}$ nur für $\lambda_1 = \lambda_2 = \dots = 0$ erfüllt wird.

2.1.5 Graphische Darstellung

Achtung: Minimierung: $-c$ Richtung! Maximierung: $+c$ Richtung!



2.1.6 Hyperebenen

Die definierenden Hyperebenen sind die Gleichungen, welche den Optimierungsbereich abgrenzen, d.h. pro Gleichung oder Bedingung (in bspw. $Ax \leq b$) entsteht eine Hyperebene. Die Notation lautet z.B. $H_1 = \{x \in \mathbb{R}^3 : x_1 + x_2 + x_3 = 2\}$. Die maximale Anzahl von Hyperebenen ist gleich der Anzahl Gleichungen.

2.1.7 Basisauswahl

Es sind bei n Dimensionen n Zeilen auszuwählen. Dabei sollen doppelte Gleichungen von Anfang an ignoriert werden. Es sind bis zu $\binom{m}{n}$ Auswahlmöglichkeiten vorhanden ($m = \#\text{Gleichungen}, n = \#\text{Variablen}$).

$$A \in \mathbb{R}^{m \times n} \quad P = \{x \in \mathbb{R}^{m \times n} : Ax \leq b\} \quad \text{Basisauswahl: } B \subseteq \{1, 2, \dots, n\} \quad A_B : \text{Basis von } A$$

Bsp.:

$$A = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ -1 & 1 \\ 0 & -1 \end{bmatrix} \quad B = \{1, 3\} \longrightarrow A_B = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$$

2.1.8 Basislösung

Berechnung der Basislösung von $P = \{x \in \mathbb{R}^{m \times n} : Ax \leq b\}$ und $B \subseteq \{1, 2, \dots, n\}$:

$$x = A_B^{-1} \cdot b_B$$

Die **Basislösung** ist **gültig** wenn $x \in P$ und entspricht dann einem Eckpunkt $v = x$ des Polyeders P !!! Um die Basislösung zu bestimmen muss $\det(\mathbf{A}_B) \neq 0$ sein (linear unabhängig).

3 Simplex-Algorithmus

1. Vorbereitung:

- Lineares Programm (LP) in Ungleichungsform bringen:

$$\max\{\mathbf{c}\mathbf{x}, \mathbf{x} \in \mathbb{R}^n : \mathbf{Ax} \leq \mathbf{b}\}$$
 (Alle Nebenbedingungen mit \leq formulieren; \mathbf{c} ist ein Zeilenvektor)
- Erste zulässige Basisauswahl $B = \{\dots\}$, für beliebigen Eckpunkt (Vertex) des Polyeders, **treffen**: Bei Vorgabe einer Ecke bilden die Ungleichungen welche diese Ecke definieren, die erste Basis.

2. Algorithmus:

(a) **Basis bestimmen:** $\mathbf{A}_B = \{\mathbf{A}\}_B$ und $\mathbf{b}_B = \{\mathbf{b}\}_B$ (B_i -te Zeilen aus \mathbf{A} und \mathbf{b} auswählen)

(b) Berechnung der **Basis-Inversen**: $\bar{\mathbf{A}} = \mathbf{A}_B^{-1}$ und der dazugehörigen **Basis-Lösung**: $\mathbf{v} = \bar{\mathbf{A}} \cdot \mathbf{b}_B$

(c) Berechnung der "reduzierten Kosten": $\mathbf{u} = \mathbf{c}\bar{\mathbf{A}}$

- $u_i < 0$: Den "negativsten" Wert u_j von \mathbf{u} und die dazugehörige Spalte $\bar{\mathbf{a}}_j$ von $\bar{\mathbf{A}}$ bestimmen.
 (Wenn die j -te Zahl in \mathbf{u} der "negativsten" Zahl entspricht so ist die j -te Spalte in $\bar{\mathbf{A}}$ die dazugehörige Spalte).
 Behalte j als Index der ursprünglichen Matrix \mathbf{A} !

- $u_i \geq 0$: Wenn \mathbf{u} keine negativen Werte aufweist, wurde das Maximum der Zielfunktion gefunden:

Argument der Maximalstelle: $\mathbf{x}^* = \mathbf{v}$

Maximalstelle: $f(\mathbf{x}^*) = \begin{cases} \mathbf{c} \cdot \mathbf{v} & \text{Wenn anfängliches Problem eine Maximierung war} \\ -\mathbf{c} \cdot \mathbf{v} & \text{Wenn anfängliches Problem eine Minimierung war} \end{cases}$

(d) **Richtungsvektor** für Bewegung auf Kante: $\mathbf{d} = -\bar{\mathbf{A}}_j$

(e) **Bewegung in Richtung von \mathbf{d}** solange alle Nebenbedingungen erfüllt.

$\mathbf{A}(\mathbf{v} + \lambda \cdot \mathbf{d}) \leq \mathbf{b} \Rightarrow \lambda^*$ (λ^* ist der Maximalwert von λ welcher alle Nebenbedingungen noch erfüllt.)

Berechnung von k : $\lambda^* = \min_k \left\{ \frac{b_i - \mathbf{a}^i \mathbf{v}}{\mathbf{a}^i \mathbf{d}} : i \in I, \mathbf{a}^i \mathbf{d} > 0 \right\}$

- $0 \leq \lambda^* < \infty$ **Weiter mit (f)**: Zielfunktion in Richtung \mathbf{d} durch Bedingung k beschränkt
- $\lambda^* \rightarrow \infty$ **Abbruch**: Zielfunktion wächst unbeschränkt in Richtung von \mathbf{d}

(f) **Basistausch**: $B'(j) = k$ bzw. $B' = B - (\{j\} \cup \{k\})$ $\mathbf{v}' = \mathbf{v} + \lambda^* \cdot \mathbf{d}$ (Die Basis an j -ter Stelle in B wird durch k ersetzt)

Setze $B := B'$, $\mathbf{v} := \mathbf{v}'$ und beginne bei Punkt (a)

3.1 Beispiel:

Das unten stehende LP-System soll mittels Simplexalgorithmus gelöst werden. Es soll im Punkt $V_0 = [0 \ 1]$ begonnen werden.

1.Durchlauf

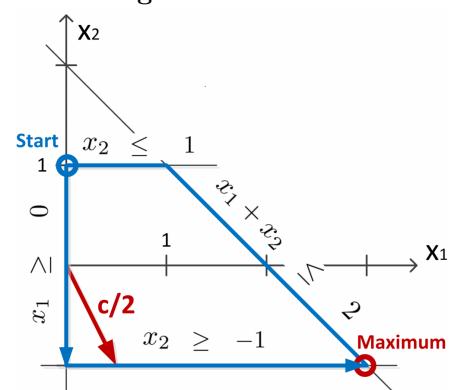
$$\begin{array}{ll} \max & \{x_1 - 2x_2\} \\ \text{max} & x_1 + x_2 \leq 2 \\ & x_1 \geq 0 \\ & x_2 \leq 1 \\ & x_2 \geq -1 \end{array}$$

In Ungleichungsform:

$$\begin{array}{ll} \max & \{x_1 - 2x_2\} \\ \text{max} & x_1 + x_2 \leq 2 \\ & -x_1 \leq 0 \\ & x_2 \leq 1 \\ & -x_2 \leq 1 \end{array}$$

Matrzenschreibweise:

$$c = [1 \quad -2] \quad A = \begin{bmatrix} 1 & 1 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \quad b = \begin{bmatrix} 2 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

Zeichnung:

$$\text{Die erste zul\"assige Basis: } B = \{2 \quad 3\} \quad \Rightarrow \quad \mathbf{A}_B = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \mathbf{b}_B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

3.1.1 Algorithmus**1.Durchlauf**

$$(a) \quad \mathbf{A}_B = \{\mathbf{A}\}_B = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \quad \mathbf{b}_B = \{\mathbf{b}\}_B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$(b) \quad \bar{\mathbf{A}} = \mathbf{A}_B^{-1} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \quad \mathbf{v} = \bar{\mathbf{A}} \cdot \mathbf{b}_B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$(c) \quad \mathbf{u} = \mathbf{c}\bar{\mathbf{A}} = [1 \quad -2] \cdot \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} = [-1 \quad -2] \quad \Rightarrow \quad \bar{\mathbf{a}}_j = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad j = 2$$

$$(d) \quad \mathbf{d} = -\bar{\mathbf{a}}_j = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

$$(e) \quad \mathbf{A}(\mathbf{v} + \lambda \cdot \mathbf{d}) \leq \mathbf{b} \quad \Rightarrow \quad \begin{bmatrix} 1 & 1 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \cdot \left(\begin{bmatrix} 0 \\ 1 \end{bmatrix} + \lambda \cdot \begin{bmatrix} 0 \\ -1 \end{bmatrix} \right) \leq \begin{bmatrix} 2 \\ 0 \\ 1 \\ 1 \end{bmatrix} \quad \Rightarrow \quad \begin{bmatrix} 1 \\ 0 \\ 1 \\ -1 \end{bmatrix} + \begin{bmatrix} -\lambda \\ 0 \\ -\lambda \\ \lambda \end{bmatrix} \leq \begin{bmatrix} 2 \\ 0 \\ 1 \\ 1 \end{bmatrix} \quad \Rightarrow \quad \lambda^* = 2 \quad k = 4$$

$$(f) \quad B'(j) = k \quad \Rightarrow \quad B' = \{2 \quad 4\} \quad \mathbf{v}' = \mathbf{v} + \lambda^* \cdot \mathbf{d} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} + 2 \cdot \begin{bmatrix} 0 \\ -1 \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

2.Durchlauf

$$(a) \quad \mathbf{A}_B = \{\mathbf{A}\}_B = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} \quad \mathbf{b}_B = \{\mathbf{b}\}_B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$(b) \quad \bar{\mathbf{A}} = \mathbf{A}_B^{-1} = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}^{-1} = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} \quad \mathbf{v} = \bar{\mathbf{A}} \cdot \mathbf{b}_B = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

$$(c) \quad \mathbf{u} = \mathbf{c}\bar{\mathbf{A}} = [1 \quad -2] \cdot \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} = [-1 \quad 2] \quad \Rightarrow \quad \bar{\mathbf{a}}_j = \begin{bmatrix} -1 \\ 0 \end{bmatrix} \quad j = 1$$

$$(d) \quad \mathbf{d} = -\bar{\mathbf{a}}_j = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$(e) \quad \mathbf{A}(\mathbf{v} + \lambda \cdot \mathbf{d}) \leq \mathbf{b} \quad \Rightarrow \quad \begin{bmatrix} 1 & 1 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \cdot \left(\begin{bmatrix} 0 \\ -1 \end{bmatrix} + \lambda \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right) \leq \begin{bmatrix} 2 \\ 0 \\ 1 \\ 1 \end{bmatrix} \quad \Rightarrow \quad \begin{bmatrix} -1 \\ 0 \\ -1 \\ 1 \end{bmatrix} + \begin{bmatrix} \lambda \\ 0 \\ 0 \\ 0 \end{bmatrix} \leq \begin{bmatrix} 2 \\ 0 \\ 1 \\ 1 \end{bmatrix} \quad \Rightarrow \quad \lambda^* = 3 \quad k = 1$$

$$(f) \quad B'(j) = k \quad \Rightarrow \quad B' = \{1 \quad 4\} \quad \mathbf{v}' = \mathbf{v} + \lambda^* \cdot \mathbf{d} = \begin{bmatrix} 0 \\ -1 \end{bmatrix} + 3 \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 3 \\ -1 \end{bmatrix}$$

3.Durchlauf

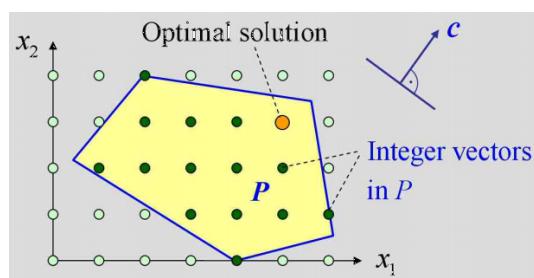
$$(a) \quad \mathbf{A}_B = \{\mathbf{A}\}_B = \begin{bmatrix} 1 & 1 \\ 0 & -1 \end{bmatrix} \quad \mathbf{b}_B = \{\mathbf{b}\}_B = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

$$(b) \quad \bar{\mathbf{A}} = \mathbf{A}_B^{-1} = \begin{bmatrix} 1 & 1 \\ 0 & -1 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 1 \\ 0 & -1 \end{bmatrix} \quad \mathbf{v} = \bar{\mathbf{A}} \cdot \mathbf{b}_B = \begin{bmatrix} 3 \\ -1 \end{bmatrix}$$

$$(c) \mathbf{u} = \mathbf{c} \overline{\mathbf{A}} = [1 \ -2] \cdot \begin{bmatrix} 1 & 1 \\ 0 & -1 \end{bmatrix} = [1 \ 3]$$

Argument der Maximalstelle: $\mathbf{x}^* = \mathbf{v} = \begin{bmatrix} 3 \\ -1 \end{bmatrix}$ Maximalstelle: $f(\mathbf{x}^*) = \mathbf{c} \cdot \mathbf{v} = [1 \ -2] \cdot \begin{bmatrix} 3 \\ -1 \end{bmatrix} = 5$

4 Ganzzahlig-lineare Programmierung Skript S. 63



Als zusätzliche Nebenbedingung zu LP dürfen Variablen nur ganzzahlig (integer, ILP) sein, das macht das Problem einiges komplizierter. Konkret sind LP polynomial lösbar, während ILP NP-vollständig (NP-complete) sind.

4.1 Relaxationen

Um Schranken (obere/untere) zu berechnen, kann der *Lösungsraum vergrößert* werden (Relaxationen). Beispiel: Der höchste Berg in Tibet ist höchstens so gross wie der höchste Berg in China (China → grösserer Lösungsraum als Tibet). In ILP wird *LP-Relaxation* verwendet, d.h. die Ganzzahligkeitsforderung wird fallen gelassen.

4.2 Branch-and-Bound Verfahren Skript S. 68

Um die optimale Lösung zu finden, wird das Lösungsgebiet aufgeteilt und jeweils sowohl eine obere als auch eine untere Schranke der Lösung gesucht. Falls die Schranken bereits eine optimale Lösung ausschliessen (wenn bei einem anderen Ast die untere Schranke höher ist als die obere beim aktuellen), kann die Suche innerhalb dieses Astes abgebrochen werden. Man spricht von *Pruning by Dominance*. Optimalität ist gefunden, wenn die obere gleich der unteren Schranke ist.

Details für ILP: Skript S. 72f.

B&B ist geeignet für beliebige Optimierungsprobleme (inkl. nichtlineare). Die Terminierung nach endlicher Anzahl Schritten ist jedoch nur bei endlichen Lösungsräumen gewährleistet.

4.2.1 Beispiel: ILP Übung 6.2

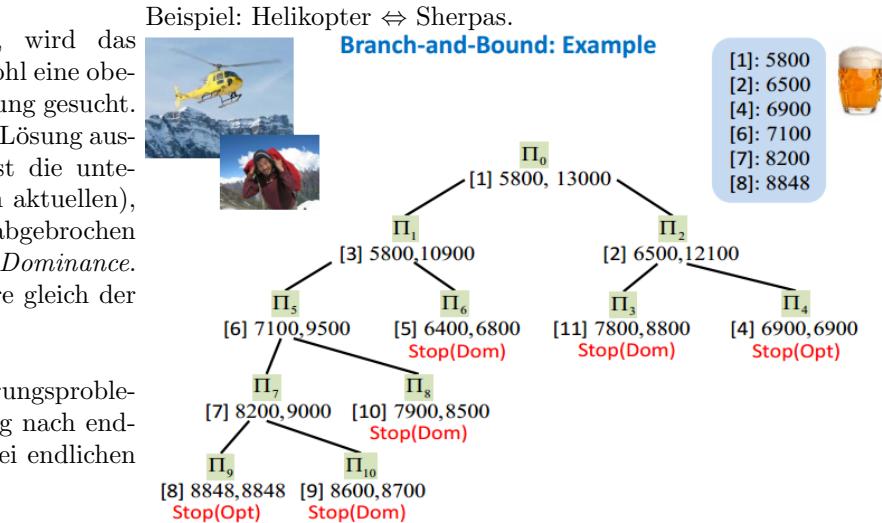
$$\begin{aligned} \max \quad & 5x_1 + 6x_2 \\ \text{3}x_1 + 6x_2 &\leq 18 \\ 9x_1 + 6x_2 &\leq 27 \\ x_1, x_2 &\geq 0, \text{ integer} \end{aligned}$$

Branching: Runden Variable mit kleinstem Index

Strategie: *Dept-First-Search*

Jeweils aufgerundeter Knoten zuerst

[Iteration]r=Knotennummer:



4.2.2 Beispiel: Branch-and-bound beim Knapsack-Problem Skript S. 73–78

Lower bound:

Lower bound ist die global gespeicherte beste Item-Auswahl, die wir bis jetzt angetroffen haben und somit Kandidat für

globales Optimum. (Vgl. 'Bar in Kathmandu')

Upper bound:

In einem beliebigen Knoten berechnet sich der upper bound wie folgt:

- Items, für die im aktuellen Knoten noch keine Entscheidung getroffen wurde, sortieren nach $\frac{Wert}{Gewicht}$ ('Wertdichte').
- Soviele (ganze) Items einpacken wie Platz dafür vorhanden ist.
- Vom nächst'dicheren' einen Bruchteil einpacken, s.d. das maximale Gewicht ausgereizt ist. (LP-Relaxation)
- Mehr Wert kann in diesem Knoten nicht erreicht werden, somit handelt es sich um den upper bound.
- Wenn der upper bound kleiner ist, als was wir (global gespeichert) schon erreicht haben, können wir abbrechen. (*pruning by dominance*)

Heuristische Lösungssuche:

Packe von den noch nicht definitiv (ab)gewählten Elementen soviele (nach Wertdichte sortierte) ein, wie es dafür Platz im Rucksack hat. (ACHTUNG: Im Wurzelknoten hat es also typischerweise zu Beginn bereits mehr als ein Element im Rucksack.)

Definition der Teilprobleme:

Um aus einem Knoten die beiden Nachfolge-Knoten zu bilden wird wie folgt vorgegangen:

- Dasjenige Element, das nur partiell Platz hatte im rechten Teilbaum definitiv wählen und im linken definitiv abwählen.
- Die heuristische Lösungssuche neu durchführen.

Strategie der Knotenwahl:

Für die Wahl des nächsten zu expandierenden Knotens gibt es verschiedene Ansätze. Wir fahren eine Depth-first-search Strategie. Dazu expandieren wir immer den rechtesten Knoten, welcher noch nicht geprunt wurde.

Arbeitsschritte:

1. Iterationsnummer inkrementieren und in eckigen Klammern notieren
2. Zu expandierenden Knoten r bestimmen (gemäss 'Strategie der Knotenwahl')
3. Vorgänger-Knoten in Klammern notieren (falls vorhanden, sonst '.)
4. Menge der definitiv abgewählten Knoten J_r^0 und definitiv gewählten Knoten J_r^1 notieren.
5. Bitvektor-Auswahl für upper bound $x_r^{UB} = x^{LP}$ notieren (enthält Bruchteil von item)
6. (Imaginären) Rucksackwert $z_r^{UB} = z^{LB}$ berechnen und auf pruning by dominance prüfen ($z_r^{UB} \leq z^{LB}$) (bester bisher erreichter Wert kann sicher nicht erreicht werden); Falls pruning by dominance weiter mit Schritt 1.
7. Bitvektor-Auswahl x_r^{LB} notieren und zugehörigen Rucksackwert z_r^{LB} berechnen
8. Auf global update für lower bound ($z_r^{LB} > z^{LB}$) prüfen, ggf. updaten.
9. Auf pruning by optimality ($z_r^{LB} = z^{UB}$) prüfen.
10. Branching ausführen (Teilprobleme bilden)
11. Weiter bei Schritt 1.

4.3 Schnittebenenverfahren (Cutting Planes) Skript S. 78

Durch Linearkombination der Bedingungen, ist eine Einschränkung des Lösungsgebiets möglich. Die Multiplikatoren werden durch "pröbeln" gefunden. Beispiel:

Ursprüngliche Bedingung 1

$$x_1 + 3x_2 \leq 6$$

$$\mid \cdot \frac{5}{8}$$

Ursprüngliche Bedingung 2

$$3x_1 + x_2 \leq 6$$

$$\mid \cdot \frac{1}{8}$$

Aufsummiert:

$$x_1 + 2x_2 \leq [4.5] \overbrace{=}^! 4$$

4.3.1 Chvátal-Rang berechnen Skript S. 91

Für rationale Polytope P existiert eine endliche Zahl k (Chvátal-Rang): Es ist die kleinste Anzahl Anwendungen des obigen Verfahrens, die nötig ist, damit das Polytop zur ganzzahligen Hülle $P_{\mathbb{Z}}$ wird.

5 Nichtlineare Optimierung

Satz von Schwarz: Skript S. 2 $\frac{\partial^2 f(x_1, x_2)}{\partial x_1 \partial x_2} = \frac{\partial^2 f(x_1, x_2)}{\partial x_2 \partial x_1}$

Bei partiellen Ableitungen spielt die Reihenfolge keine Rolle. (Bedingung: Alle partiellen Ableitungen stetig.)

5.1 Gradient Skript S. 3

Der Gradient zeigt immer in Richtung des steilsten Anstiegs der Funktion f und steht senkrecht auf den Höhenlinien.

$$\nabla f = \begin{bmatrix} f_{x_1} \\ f_{x_2} \\ \vdots \\ f_{x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$$

5.3 Gradientenverfahren Skript S. 4

Von der Stelle $\vec{x}^{(i)}$ wird in Richtung des Antigradianten geschriften.

$$\vec{x}^{(i+1)} = \vec{x}^{(i)} - \alpha_i \nabla f(\vec{x}^{(i)})$$

5.3.1 Eigenschaften

Vorteile:

- Einfach zu implementieren
- Findet optimale Lösung
- Keine gute Startnäherung nötig

Nachteile:

- Konvergiert langsam
- In der Nähe der optimalen Lösung Zick-Zack-Kurs

5.3.2 Algorithmus (zur Minimierung)

5.2 Algebraisch Extrema bestimmen

Alle Ränder müssen separat angeschaut werden!!!

(Randwerte einsetzen, dann in verbliebene Richtungen ableiten)

Sind alle partiellen Ableitungen bis zur Ordnung 2 von $f(x, y)$ stetig und gilt $f_x = 0 = f_y$ im Punkte (x_0, y_0) , so ist mit $\Delta = f_{xx} \cdot f_{yy} - f_{xy}^2$:

1.) (x_0, y_0) keine Extremalstelle, falls $\Delta < 0$

2.) (x_0, y_0) eine Extremalstelle, falls $\Delta > 0$, und zwar

- i.) ein (lokales) Minimum, wenn $f_{xx} > 0$ (resp. $f_{yy} > 0$)
- ii.) ein (lokales) Maximum, wenn $f_{xx} < 0$ (resp. $f_{yy} < 0$)

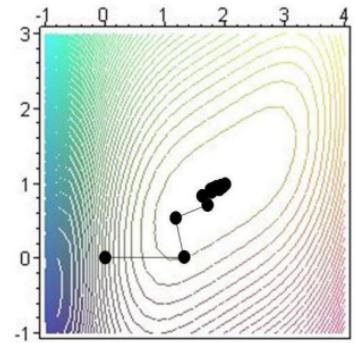
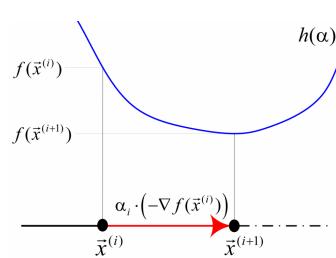
5.4 Differential-Rechnung

$$f'(x_0) = \lim_{\Delta x \rightarrow 0} \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x}$$

$$\text{Kettenregel: } (f(g(x)))' = g'(x) \cdot f'(g(x))$$

$$\text{Produktregel: } (f(x) \cdot g(x))' = f'(x) \cdot g(x) + f(x) \cdot g'(x)$$

$$\text{Quotientenregel: } \left(\frac{f(x)}{g(x)} \right)' = \frac{f'(x)g(x) - f(x)g'(x)}{g^2(x)}$$



$$1. \beta_1 = 0 \quad \beta_3 = 1 \quad h(\beta_i) = f\left(\vec{x}^{(i)} - \beta_i \nabla f(\vec{x}^{(i)})\right)$$

$$2. \text{while} \left(h(\beta_3) \geq h(\beta_1) \right) \left\{ \beta_3 = \frac{\beta_3}{2} \right\}$$

$$3. \text{Parabel konstruieren: } P(\beta) = A\beta^2 + B\beta + C \quad \text{mit} \quad \beta_1 = 0, \quad \beta_2 = \beta_3/2$$

$$\begin{bmatrix} \beta_1^2 & \beta_1 & 1 \\ \beta_2^2 & \beta_2 & 1 \\ \beta_3^2 & \beta_3 & 1 \end{bmatrix} \cdot \begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} h(\beta_1) \\ h(\beta_2) \\ h(\beta_3) \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 0 & 1 \\ \beta_3^2/4 & \beta_3/2 & 1 \\ \beta_3^2 & \beta_3 & 1 \end{bmatrix} \cdot \begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} h(0) \\ h(\beta_3/2) \\ h(\beta_3) \end{bmatrix} \Rightarrow \begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} \frac{2h(\beta_1) - 4h(\beta_2) + 2h(\beta_3)}{\beta_3^2} \\ \frac{-3h(\beta_1) + 4h(\beta_2) - h(\beta_3)}{\beta_3} \\ h(\beta_1) \end{bmatrix}$$

$$4. \text{if} \left(\beta_3 \leq -B/2A \right) \left\{ \hat{a} = \beta_3 \right\} \\ \text{else} \left\{ \hat{a} = -B/2A \right\}$$

$$5. \vec{x}^{(i+1)} = \vec{x}^{(i)} - \hat{a} \nabla f(\vec{x}^{(i)}) \Rightarrow \text{Punkt 1.}$$

5.4.1 Matlab-Implementation (nur ein Loop)

```

1 % Find betas
2 beta1 = 0;
3 beta3 = 2; % will be divided by 2 in the loop -> start value = 1
4 h_beta1 = subs(f, x, p);
5 h_beta3 = 2*h_beta1; % must be higher than h_beta1 (initial)
6 while h_beta1 <= h_beta3;
7     beta3 = beta3 / 2;

```

```

8      h_beta3 = subs(f, x, p - beta3 * gradf_numeric);
9  end;
10     beta2 = beta3/2;
11     h_beta2 = subs(f, x, p - beta2 * gradf_numeric);
12
13 % Parabola
14 C = h_beta1;
15 A = (h_beta3 - 2.0*h_beta2+C)/(2*beta2*beta2);
16 B = (h_beta2-C)/beta2 - beta2*A;
17
18 % Candidates for a
19 a1 = -B / (2*A);
20 a2 = beta3;
21
22 % Select a
23 if subs(f, x, p - beta3*gradf_numeric) <= h_beta3
24     a = a1;
25 else
26     a = a2;
27 end
28
29
30 % Calculate next step
31 p = p - a * gradf_numeric;

```

5.5 Newton-Verfahren Skript S. 7

Tangentengleichung:

$$t(x^{(i+1)}) = f(x^{(i)}) + f'(x^{(i+1)}) \cdot (x^{(i)}) = 0 \Rightarrow x^{(i+1)} = x^{(i)} - \frac{f(x^{(i)})}{f'(x^{(i)})}$$

Bestimmung der Ableitung (numerisch):

$$f'(x^{(i)}) \approx \frac{f(x^{(i)} + h) - f(x^{(i)})}{h} \quad \text{für } h \ll 1$$

Anstelle des Gradienten wird eine Tangente an den Arbeitspunkt gelegt.

$$\vec{x}^{(i+1)} = \vec{x}^{(i)} - \left(\mathbf{H}(\vec{x}^{(i)}) \right)^{-1} \nabla f(\vec{x}^{(i)})$$

mit der Hesse-Matrix

$$\mathbf{H}(\vec{x}) = \left(\frac{\partial^2 f}{\partial x_i \partial x_j}(x) \right)_{i,j=1,\dots,n} = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1}(x) & \frac{\partial^2 f}{\partial x_1 \partial x_2}(x) & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n}(x) \\ \frac{\partial^2 f}{\partial x_2 \partial x_1}(x) & \frac{\partial^2 f}{\partial x_2 \partial x_2}(x) & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n}(x) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1}(x) & \frac{\partial^2 f}{\partial x_n \partial x_2}(x) & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n}(x) \end{pmatrix}$$

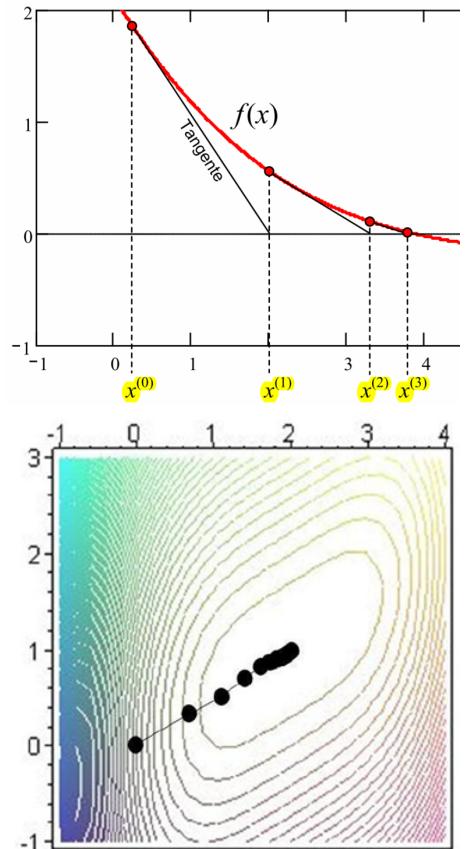
(4 Funktionsaufrufe für Diagonalelemente, sonst 3 Funktionsaufrufe)

Vorteile:

- Gutes Konvergenzverhalten

Nachteile:

- Hesse-Matrix benötigt zweite Ableitung (numerisch aufwendig)
- Inversion der Matrix skaliert schlecht ($O(n^3)$)



5.6 Quasi-Newton-Verfahren Skript S. 11

Idee: Die Ableitung des Newton-Verfahrens wird durch die Sekantensteigung ersetzt.

$$F'(x^{(i+1)}) \approx \frac{F(x^{(i+1)}) - F(x^{(i)})}{x^{(i+1)} - x^{(i)}}$$

Broyden Quasi-Newton Verfahren:

Initialisierung

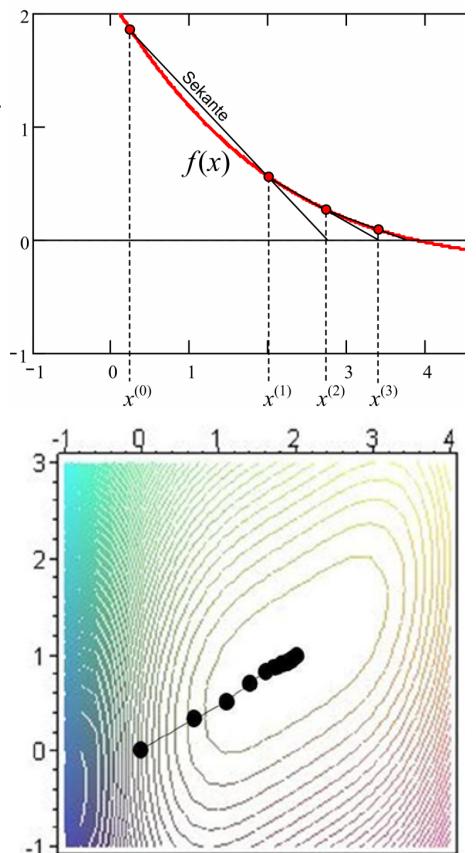
- (a) Hesse-Matrix bestimmen: $\mathbf{H}(\vec{x}^{(0)})$
Gradient bestimmen: $\nabla f(\vec{x}^{(0)})$

$$(b) \quad \mathbf{A}_0^{-1} = (\mathbf{H}(\vec{x}^{(0)}))^{-1}$$

$$(c) \quad \vec{x}^{(1)} = \vec{x}^{(0)} - \mathbf{A}_0^{-1} \cdot \nabla f(\vec{x}^{(0)})$$

Loop

1. $\vec{y}_i = \nabla f(\vec{x}^{(i)}) - \nabla f(\vec{x}^{(i-1)}) \quad \vec{s}_i = \vec{x}^{(i)} - \vec{x}^{(i-1)} \quad i \geq 1$
2. $\mathbf{A}_i^{-1} = \mathbf{A}_{i-1}^{-1} - \frac{(\mathbf{A}_{i-1}^{-1} \cdot \vec{y}_i - \vec{s}_i) \vec{s}_i^T \cdot \mathbf{A}_{i-1}^{-1}}{\vec{s}_i^T \cdot \mathbf{A}_{i-1}^{-1} \cdot \vec{y}_i} \quad i \geq 1$
3. $\vec{x}^{(i+1)} = \vec{x}^{(i)} - \mathbf{A}_i^{-1} \cdot \nabla f(\vec{x}^{(i)}) \quad \text{zu Punkt 1.}$



Um die Berechnung der Ableitungen zu umgehen, werden diese durch finite Differenzen approximiert.

$$\frac{\partial^2 f}{\partial x_i^2} = \frac{1}{\epsilon^2} (f(x_1, x_2, \dots, x_i + \epsilon, \dots, x_n) - 2f(x_1, x_2, \dots, x_i, \dots, x_n) + f(x_1, x_2, \dots, x_i - \epsilon, \dots, x_n))$$

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{1}{4\epsilon^2} \left(f(x_1, x_2, \dots, x_i + \epsilon, \dots, x_k + \epsilon, \dots, x_n) - f(x_1, x_2, \dots, x_i - \epsilon, \dots, x_k + \epsilon, \dots, x_n) - f(x_1, x_2, \dots, x_i + \epsilon, \dots, x_k - \epsilon, \dots, x_n) + f(x_1, x_2, \dots, x_i - \epsilon, \dots, x_k - \epsilon, \dots, x_n) \right)$$

5.7 Konvergenzbeschleunigung Skript S. 17

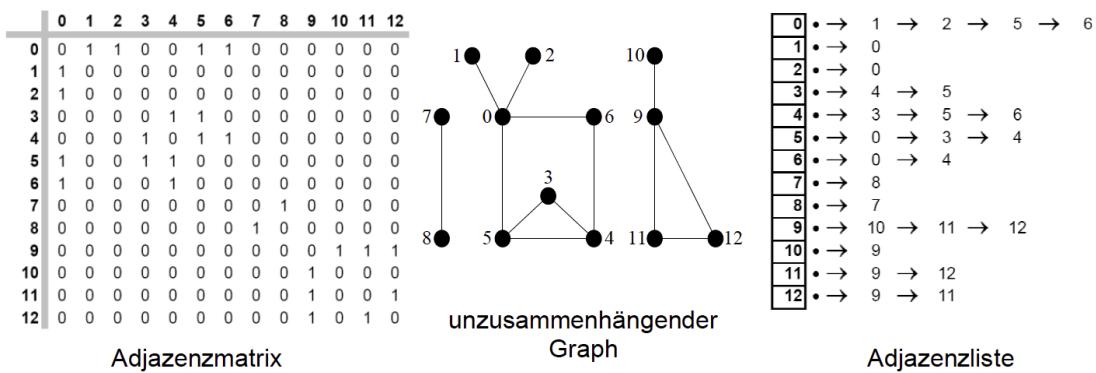
Idee: Mit der sogenannten Aitken'sche ∇^2 -Methode wird aus der Folge $\{x_i\}$ die die schneller konvergierende Folge $\{\hat{x}_i\}$ berechnet.

Aitken ∇^2 -Methode:	$\hat{x} = x_i - \frac{(x_{i+1} - x_i)^2}{x_{i+2} - 2x_{i+1} + x_i} \quad i \geq 0$	(Formel dimensionsweise anwenden, für x_1, x_2, \dots)
-----------------------------	---	---

6 Graphen und Netzwerke

Exakter Algorithmus	Findet optimale Lösung
Heuristik	Findet gute, aber normalerweise nicht optimale Lösung
Greedy Algorithmus	Trifft zu jedem Zeitpunkt aktuell beste Wahl, d.h. lokales Optimum; müssen nicht zu globalen Optimum führen
Ungerichteter Graph	Graph mit endlicher Menge Ecken V und Kanten $v, w \in E$
Einfacher Graph	Keine Schlingen, keine parallelen Kanten
Schlinge	Kante, wo Anfang- gleich Endecke ist (vv)
Pfad	Folge von Ecken v_0, v_1, \dots, v_k
Kreis	Pfad, wo die Anfangs- und Endecke gleich sind: $v_0 = v_k$
Azyklischer Graph, Wald	Graph, der keinen Kreis enthält
Baum	Azyklischer, zusammenhängender Graph
Bipartiter Graph	Die Ecken können in zwei Teilmengen aufgeteilt werden, so dass jede Kante je eine Ecke in beiden Teilmengen hat Skript S. 10
Eulerkreis	Kreis, der jede Kante des Graphen genau ein Mal enthält
Hamiltonkreis	Kreis, der alle Ecken genau ein Mal besucht und zudem in der Startecke startet und endet
Euler-/Hamiltongraph	Wenn Graph einen Euler- oder Hamiltonkreis enthält

6.1 Repräsentation von Graphen Skript S. 3



Adjazenzmatrix		Adjazenzliste
$O(n^2)$	Speicherplatz	$O(n+m)$
$O(1)$	Test ob $(v_i, v_k) \in E$	$O(n)$
$O(n^2)$	Traversieren aller Kanten	$O(n+m)$
$O(n^2)$	Weg von v_i nach v_k	$O(n+m)$

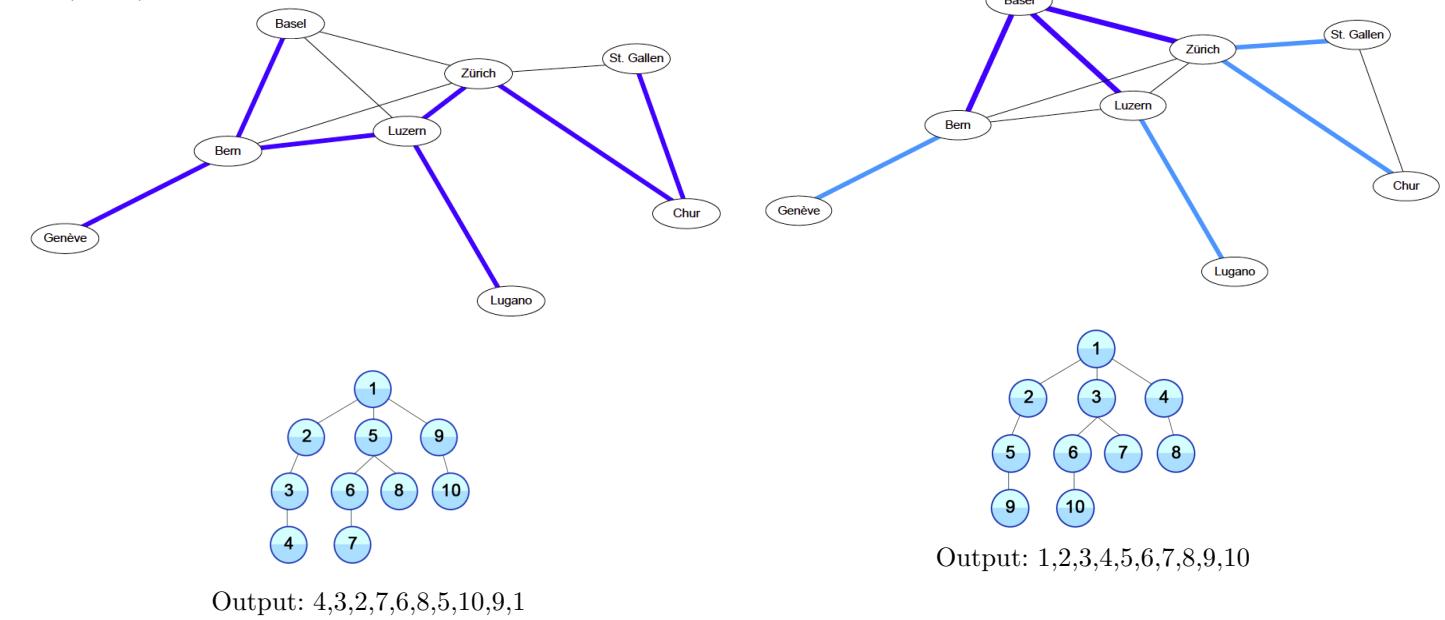
6.2 Entscheidungsbaum-Verfahren Skript S. 4

Das Travelling-Salesman-Problem (Skript S. 31), wo jede Stadt genau ein Mal besucht werden darf, ist lösbar mit dem Branch-And-Bound Verfahren.

6.3 Traversieren von Graphen Skript S. 9

6.3.1 Tiefensuche (Depth-First Search, DFS)

Ausgehend von einer Ecke, wird immer weiter zur nächsten Ecke durch den Baum "traversiert", bis das Ende erreicht ist. Von einer Ecke werden alle benachbarten Ecken aufgesucht und dann wird von der nächsten unbesuchten Ecke weitertraversiert, bis alle Ecken besucht sind. Dies ist eine Last-In-First-Out (LIFO) Strategie.



6.4 Minimum Spanning Tree (MST)

MST = Aufstellen eines Minimalen Baums: Summe aller Kantengewichte ist minimal unter allen verbundenen Knoten.

Prim-Algorithmus	Läuft über Ecken, ist greedy	$O(n^2)$
Kruskal-Algorithmus	Läuft über Kanten, ist greedy	$O(m \log(m))$
MST-Heuristik zur Lösung des TSP	Von bestehendem MST aus, wird in beliebige Richtung iteriert	$O(n) ??$

6.4.1 Prim-Algorithmus

Setzt Ecken ($O(n^2)$) Zwischengraph ist immer zusammenhängend.

1. Beliebige Startecke v_0 für den MST wählen
2. Ecke v_i mit kleinstem Abstand zu einer Ecke im MST hinzufügen
3. Punkt 2 wiederholen bis alle Ecken im MST

6.4.2 Kruskal-Algorithmus

Setzt Kanten ($O(m \log(m))$) Zwischengraph ist allenfalls unzusammenhängend.

1. Kante mit kleinstem Gewicht in den MST aufnehmen
2. Kante e_i mit kleinstem Gewicht, die zwei Ecken verbindet die noch nicht durch den MST verbunden sind, im MST aufnehmen
3. zu 2 bis alle Ecken verbunden sind

6.4.3 MST-Heuristik zur Lösung des TSP

Generiert Hamiltonkreis (Lösung des TSP) aus MST (Lösung muss nicht optimal sein!, Lösung ist abhängig vom Startpunkt)

1. Jede Kante im MST verdoppeln (Eulergraphen)
2. Eulerkreis auswählen (durchläuft alle Kanten)
3. Eulerkreis durchlaufen und dabei alle bereits besuchten Knoten überspringen (Hamiltonkreis)

6.5 Minimaler Weg

6.5.1 Probleme

Source-sink shortest path	Kürzesten Weg zwischen einer Ecke (Quelle/source) und einer andern Ecke (Senke/sink)
Single source shortest paths	Kürzeste Wege von einer Ecke (Quelle/source) zu allen anderen Ecken
All pairs shortest paths	Kürzeste Wege zwischen allen Eckenpaaren

6.5.2 Algorithmen

Algorithmus	Löst	Bemerkungen	Compl.
Dijkstra <small>Skript S. 14</small>	Source-sink shortest path, single source shortest path	Kantengewichte $\omega(v_i v_j) \geq 0$; Greedy; Loop über alle bereits besuchten Ecken: Suche minimale Distanz zu nächster Ecke. Der Weg, der am Schnellsten am Ziel ist, gewinnt.	$O(n^2)$
Bellman-Ford	Source-sink shortest path, single source shortest path	Auch negative Kantengewichte möglich; n Knoten, m Kanten	$O(n \cdot m)$
A*	Single source shortest path	Heuristische Methode des Dijkstra, beschleunigte Laufzeit, findet optimale Lösung, benötigt viel(!) Speicher	?
Floyd-Warshall <small>Skript S. 16</small>	All pairs shortest paths	Gerichtete Kreise möglich, Kantengewichte $\omega(v_i v_j) \geq 0$	$O(n^3)$

6.5.3 Dijkstra

Sucht die kürzeste Verbindung zwischen Knoten v_0 und v_n Kantengewicht müssen positiv sein! (sonst kann es zu nicht-optimalen Lösungen führen, da ein nicht beachteter Umweg schlussendlich kürzer gewesen wäre) Die Menge V enthält alle Ecken die bereits erreicht wurden. l_i ist der kürzeste Weg von der Ecke v_0 zur Ecke v_i .

1. Beliebige Startecke v_0 wählen und in die Ecken-Menge V aufnehmen
2. Aus den von V direkt erreichbaren Ecken diejenige zu V hinzufügen, die zu v_0 die kürzeste Distanz hat
3. $l_j = l_i + w$ (j ist die neue Ecke und i die Ecke, die bereits in V ist, w ist die Distanz zwischen i und j)
4. Gehe zu 2 bis die Ecke v_n zu V hinzugefügt wird

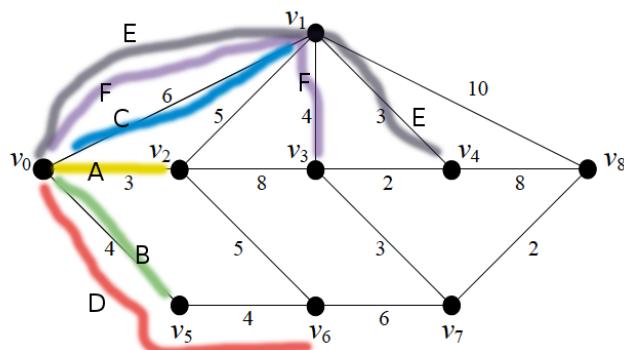


Illustration des Dijkstra–Algorithmus an einem Beispiel

	$V_0 = \{ v_0 \}$	$E_0 = \{ \}$	$l(v_0) = 0$
A	$e_1 = v_0 v_2$	$V_1 = \{ v_0, v_2 \}$	$E_1 = \{ v_0 v_2 \}$
B	$e_2 = v_0 v_5$	$V_2 = \{ v_0, v_2, v_5 \}$	$E_2 = \{ v_0 v_2, v_0 v_5 \}$
C	$e_3 = v_0 v_1$	$V_3 = \{ v_0, v_2, v_5, v_1 \}$	$E_3 = \{ v_0 v_2, v_0 v_5, v_0 v_1 \}$
D	$e_4 = v_5 v_6$	$V_4 = \{ v_0, v_2, v_5, v_1, v_6 \}$	$E_4 = \{ v_0 v_2, v_0 v_5, v_0 v_1, v_5 v_6 \}$
E	$e_5 = v_1 v_4$	$V_5 = \{ v_0, v_2, v_5, v_1, v_6, v_4 \}$	$E_5 = \{ v_0 v_2, v_0 v_5, v_0 v_1, v_5 v_6, v_1 v_4 \}$
F	$e_6 = v_1 v_3$ $e_7 = v_3 v_7$ $e_8 = v_7 v_8$	$V_6 = \{ v_0, v_2, v_5, v_1, v_6, v_4, v_3 \}$ $V_7 = \{ v_0, v_2, v_5, v_1, v_6, v_4, v_3, v_7 \}$ $V_8 = \{ v_0, v_2, v_5, v_1, v_6, v_4, v_3, v_7, v_8 \}$	$E_6 = \{ v_0 v_2, v_0 v_5, v_0 v_1, v_5 v_6, v_1 v_4, v_1 v_3 \}$ $E_7 = \{ v_0 v_2, v_0 v_5, v_0 v_1, v_5 v_6, v_1 v_4, v_1 v_3, v_3 v_7 \}$ $E_8 = \{ v_0 v_2, v_0 v_5, v_0 v_1, v_5 v_6, v_1 v_4, v_1 v_3, v_3 v_7, v_3 v_8 \}$

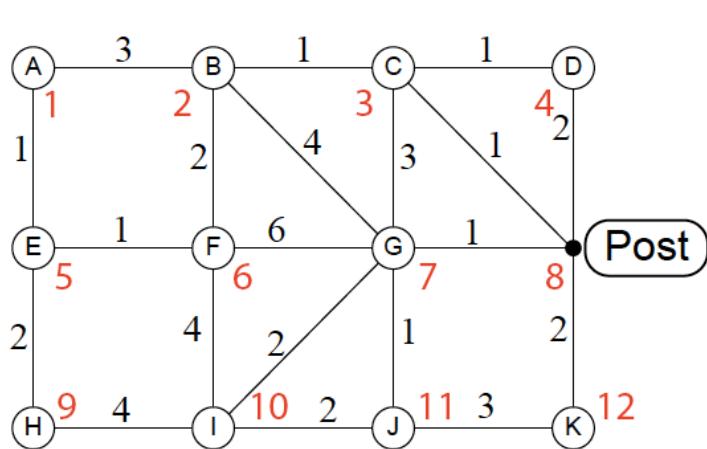
6.5.4 A*

Der A*-Algo. untersucht immer die Knoten zuerst, die wahrscheinlich schnell zum Ziel führen. Diese Wahrscheinlichkeit wird ermittelt, in dem jedem bekannten Knoten x ein Wert $f(x)$ zugewiesen wird. $f(x)$ beschreibt die Distanz vom Start zum Ziel über den verwendeten Knoten ist. Der Knoten mit dem kleinsten $f(x)$ wird als nächster untersucht.

$$f(x) = g(x) + h(x) \quad g(x) : \text{Weg vom Startknoten bis zu } x \quad h(x) : \text{Weg von } x \text{ zum Zielknoten (Heuristik)}$$

Eine Heuristik für die Verbindung von Städten wäre z.B. die Luftlinie. Für die Durchführung des Algos wird eine **Closed-List** (untersuchte Knoten) und eine **Open-List** (gefundene, aber noch nicht untersuchte Knoten) verwendet. **Gefundene Knoten** sind, die die eine Verbindung mit einem Knoten der Closed-List haben.

Bsp.: Gesucht ist der kürzeste Weg von *Post* nach *A*. Bei diesem Beispiel wird für die Distanzheuristik $h(x)$ zwischen den Knoten v_i und v_j der Betrag $h(x) = |i - j|$ verwendet ((rot) $A = 1$, $B = 2$, etc.).



$l(x)$	closed list	open list	$g(x) + h(x)$
0	Post	C D G K	1 + 2 2 + 3 1 + 6 2 + 11
0	Post	B C D G K	2 + 1 2 + 3 1 + 6 2 + 11
0	Post	A C D F G K	5 + 0 2 + 3 4 + 2 1 + 6 2 + 11

Der Minimale Weg von der Post zum Knoten A beträgt 5 und kann rückwärts rekonstruiert werden:
 $A \rightarrow B \rightarrow C \rightarrow Post$

6.5.5 Floyd-Warshall

Alle Kreise müssen ein nicht negatives Gewicht haben! Der Algorithmus ist konzipiert für gerichtete Graphen, funktioniert auch für ungerichtete (beide Richtungen gleiches Gewicht).

$\text{shortestPath}(i, j, k)$ gibt den kürzesten weg vom Knoten v_i nach v_j , wobei der Weg über die ersten k Knoten führen kann (v_0, \dots, v_{k-1}).

Die Funktion kann rekursiv implementiert werden:

$$\text{shortestPath}(i, j, k+1) = \min \begin{cases} \text{shortestPath}(i, j, k) & \text{Weg bereits bekannt (führt über } v_0, \dots, v_{k-1}\text{)} \\ \text{shortestPath}(i, k, k) + \text{shortestPath}(k, j, k) & \text{Weg führt über } k \text{ und von dort nach } j \end{cases}$$

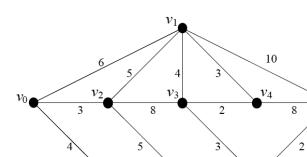
Hat ein Graph keine Kante von v_i nach v_j dann wird das Gewicht auf ∞ gesetzt!

Das Resultat des Algos ist eine Matrix, in der die kürzesten Distanzen zwischen allen Knoten angegeben ist.

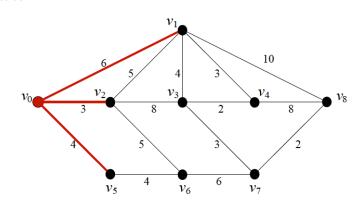
1. Init: setze $\text{shortestPath}(i, j, 0)$ (Pfade von v_i nach v_j ohne Zwischenknoten)
2. entwickle $\text{shortestPath}(i, j, k+1)$ für $k = 0, 1, \dots, n-1$ rekursive (überschreiben der Elemente bei kürzerem Weg)
3. Matrix enthält alle kürzesten Wege

Bsp.: (nichts eingetragen = ∞)

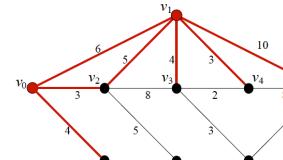
$\text{shortestPath}(i, j, 0)$								
0	1	2	3	4	5	6	7	8
0	6	3		4				
1	6	0	5	4	3			10
2	3	5	0	8				
3	4	8	0	2				3
4	3	2	0					8
5	5			0	4			
6				4	0	6		
7				3	6	0	2	
8	10			8	2	0		



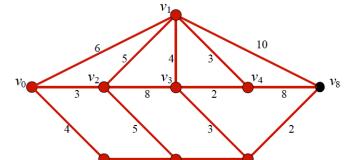
$\text{shortestPath}(i, j, 1)$									über Knoten v_0
0	1	2	3	4	5	6	7	8	
0	0	6	3		4				
1	6	0	5	4	3	10			
2	3	5	0	8	7	5			
3	4	8	0	2					3
4	3	2	0						8
5	4	10	7		0	4			
6				5	4	0	6		
7				3	6	0	2		
8	10			8	2	0			



	0	1	2	3	4	5	6	7	8
0	0	6	3	10	9	4			16
1	6	0	5	4	3	10			10
2	3	5	0	8	8	7	5		15
3	10	4	8	0	2	14	3	14	
4	9	3	8	2	0	13	8		
5	4	10	7	14	13	0	4	20	
6	5		3	4	0	6			
7			6	0	2				
8	16	10	15	14	8	20	2	0	

über Knoten v_0 und v_1 

	0	1	2	3	4	5	6	7	8
0	0	6	3	10	9	4	8	13	15
1	6	0	5	4	3	10	7		9
2	3	5	0	8	8	7	5	11	13
3	10	4	8	0	2	13	9	3	5
4	9	3	8	2	0	13	11	5	7
5	4	10	7	13	13	0	4	10	12
6	8	10	5	9	11	4	0	6	8
7	13	7	11	3	5	10	6	0	2
8	15	9	13	5	7	12	8	2	0

über v_0 bis v_7 (k von 0 bis $n-1$)

Aus der resultierenden Matrix kann nicht nur der kürzeste sondern auch der längste Pfad entnommen werden. Dieser wird auch Durchmesser des Graphen genannt.

7 Netzwerke

Jede Kante vw hat jetzt auch eine Kapazität $c(vw)$, welche eine Obergrenze für den Fluss $f(vw)$ darstellt. Zusätzlich können den Kanten noch Kosten $k(vw)$ angefügt werden.

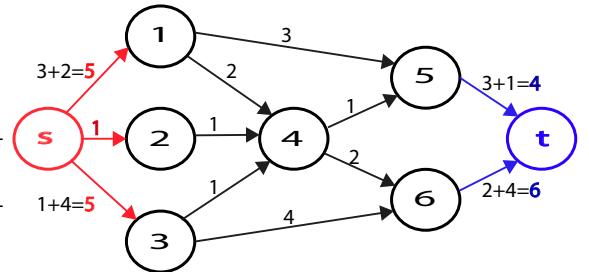
- | | |
|--------------------|---|
| Verteilungsproblem | Objekte von einem oder mehreren Orten (sources) an eines oder mehrere Ziele (sinks) zu verteilen |
| Matching-Probleme | Zuordnung von Mengen, kann auch als bipartite Graphen interpretiert werden (bspw. Partnervermittlung) |
| Schnitt-Probleme | Kanten entferne, um Netzwerk zu zerlegen (bspw. max. Anzahl Unterbrüche in Kommunikationsnetz) |

7.1 Reduktion auf Maxflow-Problem

Viele bestehenden Probleme können auf Maxflow-Probleme reduziert werden.

Mehrere Quellen / Mehrere Senken (Übung II.5/1):

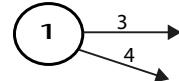
1. Neue Quelle s einführen (Kantenkapazitäten c_s entsprechen der Summe der wegführenden Kapazitäten)
2. Neue Senke t einführen (Kantenkapazitäten c_t entsprechen der Summe der hinführenden Kapazitäten)



Einschränkung durch Maximalkapazität:

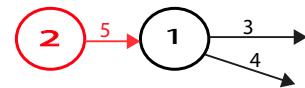
Begrenzungen durch Einführen von zusätzlichen Knoten.

Output von Knoten 1 auf 5 beschränkt!



Begrenzte Output-Kapazität:

Neuen Knoten einführen (Kantenkapazitäten c entspricht der beschränkten Output-Kapazitäten)

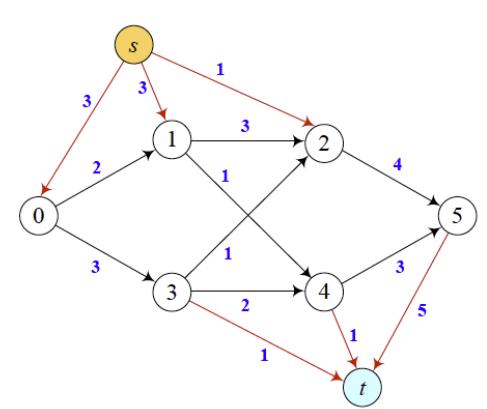
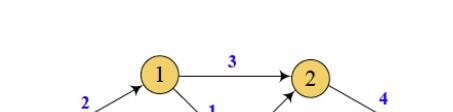


Logistik-Problem (Übung II.5/2):

Lieferanten 0, 1, 2 haben eine Angebotskapazität

Bezüger 3, 4, 5 haben eine Nachfrage
neu eingefügte Quelle s und Senke t versorgen die Anbieter bzw. Bezüger mit den Angebot- bzw. Nachfragekapazitäten

Ecke	0	1	2	3	4	5
Angebot	3	3	1	1	1	5
Nachfrage						



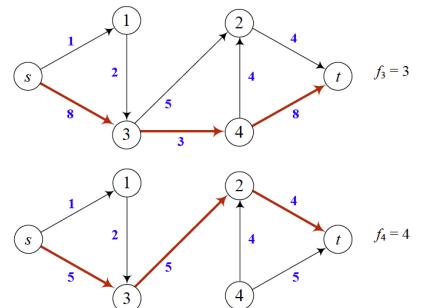
7.2 Maxflow Problem

Gesucht wird der maximal mögliche Fluss zwischen einer Quelle s zu einer Senke t (st -Netzwerk). Für Komplexität: $m =$ Anzahl Kanten, $f =$ maximaler Fluss von s nach t .

Algorithmus	Beschreibung	Compl.
Ford-Fulkerson	Kapazitäten müssen ganzzahlig sein! Pfade werden ausprobiert und jeweils der maximale Fluss durch diesen Pfad als (fiktiven) Rückwärtsfluss auf allen Kanten aufgetragen.	$O(m \cdot f)$
Skript S. 22		
Edmonds-Karp	Gleich wie Ford-Fulkerson, aber es wird eine Breitensuche BFS angewendet.	$O(n \cdot m^2)$
Skript S. 24		
Max-Flow Min-Cut	Der maximale Fluss in einem st -Netzwerk ist gleich dem minimalen Fluss über alle möglichen st -Schnitte.	-

7.2.1 Augmenting-Path Methode

1. Beliebigen Pfad von s nach t wählen
2. Begrenzenden Teil-Pfad ermitteln; Kleinste Kapazität f_i notieren
3. Abziehen von f_i von den Kapazitäten des gewählten Pfades f_j ($f_j - f_i$)
4. Wenn ein weiterer Pfad von s nach t führt gehe zu Punkt 1, andernfalls zu 5
5. Der Gesamtfluss ist die Summe aller begrenzten Flüsse $f_{tot} = \sum_{i=1} f_i$



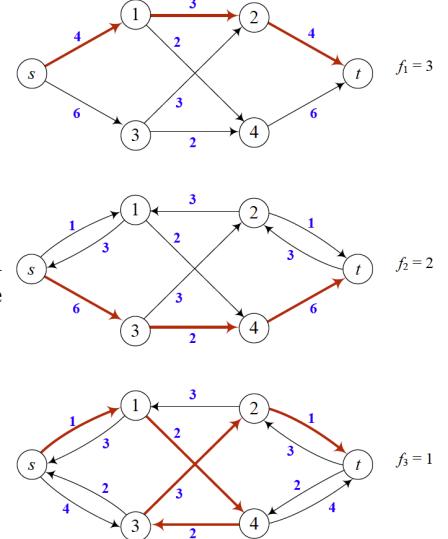
Vorgehen ist problematisch, da es nicht möglich ist, einmal gewählte Flüsse wieder abzuwählen. (Greedy Verhalten)

7.2.2 Ford-Fulkerson

Vorwärtspfad: Gibt an, um wieviel der Durchfluss erhöht werden kann.

Rückwärtspfad: Gibt an, um wieviel der Durchfluss verringert werden kann.

1. Beliebigen Pfad von s nach t wählen
2. Begrenzenden Teil-Pfad ermitteln; Kleinste Kapazität f_i notieren
3. Abziehen von f_i von den Vorwärtskanten f_j und hinzufügen bei den Rückwärtskanten r_j des gewählten Pfades f_j . Existiert keine Rückwärtskante muss diese eingefügt werden ($f_j - f_i$ und $r_j + f_i$)
4. Wenn ein weiterer Pfad von s nach t führt gehe zu Punkt 1, andernfalls zu 5
5. Der Gesamtfluss ist die Summe aller begrenzenden Flüsse $f_{tot} = \sum_{i=1} f_i$



7.2.3 Max-Flow Min-Cut

Theorem: Max-Flow ist gleich dem minimalen Fluss über alle möglichen ST-Cuts!

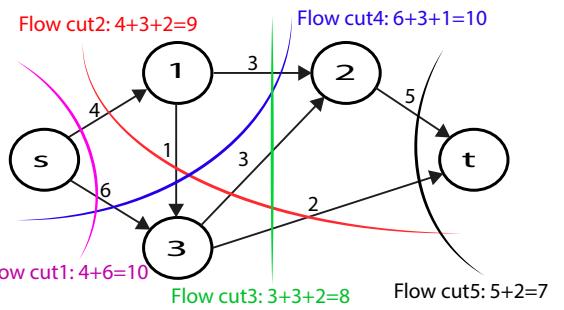
Source Sink Cuts (ST-Cuts):

ST-Cut: Aufteilen der Ecken in eine Menge S (enthält Quelle s) und eine Menge T (enthält Senke t); $2^{|E|}$ Möglichkeiten

Flow über ST-Cut: Summe aller Kapazitäten der Kanten, welche in S beginnen und in T enden!

Max-Flow Min-Cut: im Bsp. ist der minimale Fluss über alle Cuts (Cut 5) gleich 7, was dem Maximalen Fluss des ST-Netzwerks entspricht.

Min-Cut entspricht dem Nadelöhr, Max-Flow kann nur so viel sein, wie durchs Nadelöhr passt.



7.3 Maxflow Mincost Skript S. 28

Die Lösung des Max Flow Problems ist nicht eindeutig, darum können weitere Kriterien hinzugefügt werden.

Tipps:

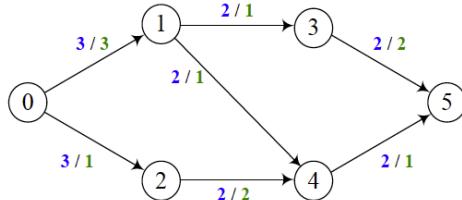
- Große Kreise zeichnen, damit Pfeilrichtungen eindeutig erkennbar.
- Zweige mit geringen Transportkosten zuerst invertieren, um mit grosser Wahrscheinlichkeit negative Kreise zu unterbinden.

Vorgehen:

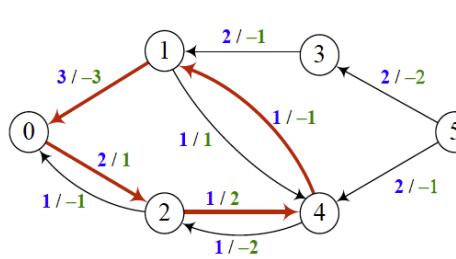
1. Ford-Fulkerson anwenden (f_{tot} bleibt konstant!)
2. Restnetzwerk durch Kantenkosten k erweitern (Vorwärtskante: $+k$, Rückwärtskante: $-k$)
3. Wenn Kreis mit negativen Kosten vorhanden gehe zu 4, andernfalls zu 5
4. Fluss des neg. Kreises um die min. Kantenkapazität c_{min} erhöhen (in Gegenrichtung!), gehe zu 3
5. Gesamtfluss (max): f_{tot} aus Ford-Fulkerson
6. Gesamtkosten (min): Summe aller Rückwärtskantenkosten (RK) multipliziert mit der entsprechenden Kantenkapazität:

$$k_{tot} = \sum_{i \in \text{alle RK}} c_i \cdot |k_i|$$

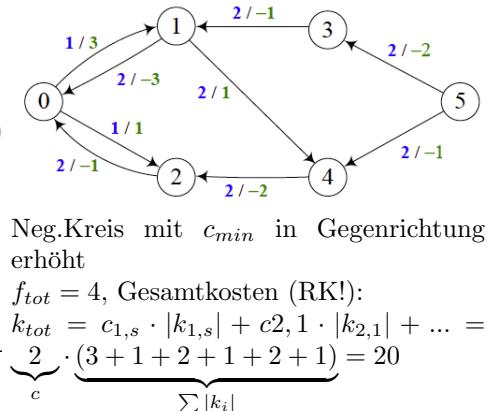
Bsp.:



Ausgangsnetzwerk mit Kapazitäten und Kosten



Restnetzwerk mit $f_{tot} = 4$, $k_{tot} = 21$,
Kreis mit neg. Kosten (rot,
 $1 + 2 - 1 - 3 = -1$) mit min. Kantenkapazität $c_{min} = 1$



8 Meta-Heuristiken

8.1 Hill-Climbing Skript S. 2

Konvergiert meist in lokales Optimum, Konvergenz sehr langsam., Wahl eines Zufallsschritts ist schwierig. Zielfunktion muss oft berechnet werden.

Algorithmus, um Zielfunktion $f(\vec{x})$ zu maximieren:

1. Start: Wahl von \vec{x}^{alt}
2. Neue Wahl \vec{x}^{neu} „in der Nähe“ von \vec{x}^{alt} (zuällig)
3. Wenn $f(\vec{x}^{neu}) \geq f(\vec{x}^{alt})$, setze $\vec{x}^{alt} = \vec{x}^{neu}$ und gehe zu Schritt 2

8.2 Tabu-Search Skript S. 3

Grundidee: Keine / möglichst wenige Schritte, welche die Wirkung früherer Schritte rückgängig machen.

Etwas schnellere Konvergenz ggü. Hill-Climbing, allerdings aufwendiges Mitführen und Aktualisieren der Tabu-Liste sowie Schwierigkeit, um Länge der Tabu-Liste zu finden. Zielfunktion muss oft berechnet werden.

Algorithmus, um Zielfunktion $f(\vec{x})$ zu maximieren:

1. Start: Wahl von \vec{x}^{alt}
2. Neue Wahl \vec{x}^{neu} „in der Nähe“ von \vec{x}^{alt} , welche mit einem erlaubten Schritt $m_i \notin T$ eine möglichst gute Verbesserung (oder geringste Verschlechterung bringt).

3. Einfügen des komplementären Schritts \bar{m}_i in Tabu-Liste T .
4. Schritte 2 und 3 wiederholen, bis vorgegebenes Abbruchkriterium erreicht ist.

8.3 Simulated Annealing Skript S. 5

Grundidee: Gegenüber dem Tabu-Search wird eine Verschlechterung der Lösung mit einer gewissen Wahrscheinlichkeit akzeptiert. So kann ein lokales Optimum wieder verlassen werden.

1. Start: Wahl von \vec{x}^{alt}
2. Neue Wahl \vec{x}^{neu} „in der Nähe“ von \vec{x}^{alt}
3. Wenn $f(\vec{x}^{neu}) \geq f(\vec{x}^{alt})$ oder mit einer Wahrscheinlichkeit von $p = e^{\frac{-\Delta E}{T}}$, setze $\vec{x}^{neu} = \vec{x}^{alt}$.
Metropolis Regel: Zufallsvariable $z \in [0, 1]$. Wenn $z \leq p$ dann wird der schlechtere Wert genommen $\vec{x}^{neu} = \vec{x}^{alt}$
4. Schritte 2 und 3 wiederholen, bis vorgegebenes Abbruchkriterium erreicht ist.

Wahl der Parameter:

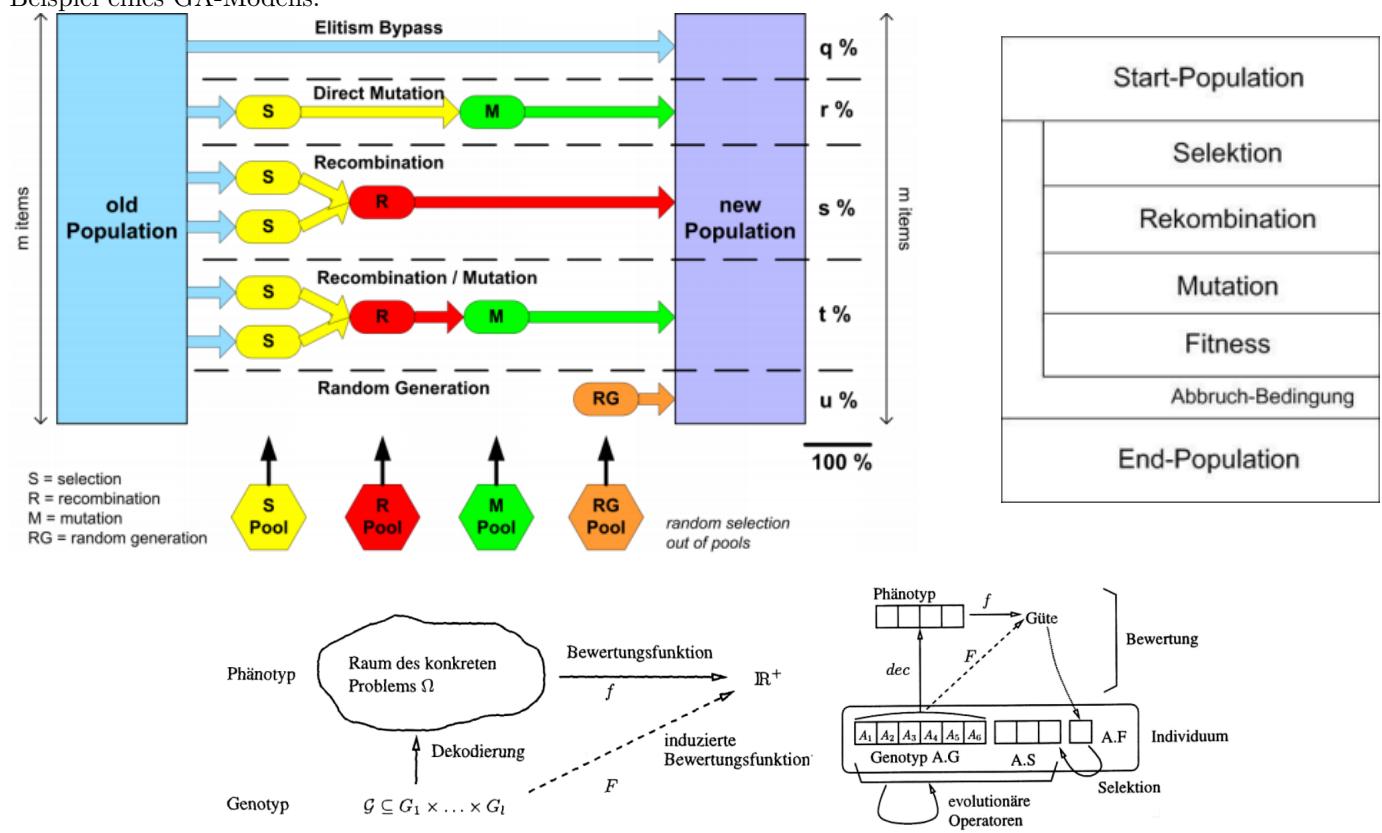
- ΔE : Wenn Minimum: $\Delta E = f(\vec{x}^{neu}) - f(\vec{x}^{alt})$; wenn Maximum: $\Delta E = f(\vec{x}^{alt}) - f(\vec{x}^{neu})$
- Veränderung der Temperatur $T(i)$ (Kühlschema): Konstant, arithmetisch (bei jedem Schritt wird um gleichen Betrag verkleinert), geometrisch (bei jedem Schritt wird um gleichen Faktor verkleinert), stufenweise (diskrete Sprünge)
- Anfangstemperatur $T(0)$: Z.B. mittlere Änderung der Zielfunktion über einige zufällig gewählte Schritte.

8.4 Genetische Algorithmen (GA) Skript S. 8

Löst alle Klassen von schwierigen Optimierungsproblemen: Parameteroptimierungen, kombinatorische Probleme (e.g. Travelling Salesman), Subset-Selektion (Stichwort Data-Mining). Nachteile sind die Erfahrung und das Geschick, das nötig ist für deren Entwicklung.

Grundprinzip: Durch *Selektion* (Fitness/objective function), *Rekombination* (Kombination des genetischen Materials der Eltern) und *Mutation* (Veränderung der Erbinformation) können Funktionen optimiert werden. Dabei wird iterativ von einer „alten“ Population durch verschiedene Operationen eine neue Population generiert, welche im Normalfall gleich gross sein soll.

Beispiel eines GA-Modells:



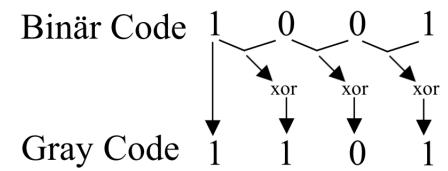
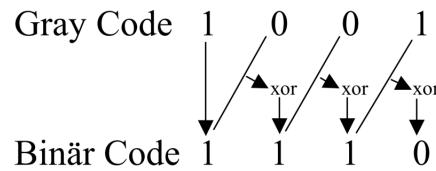
8.4.1 Codierung der Parameter Skript S. 11

Am besten werden die Parameter *Gray*-codiert. Damit wird erreicht, dass für jeden möglichen Parameter-Wert zwei Ein-Bit-Mutationen existieren, die den nächstgrößeren und -kleineren Parameter-Wert bilden.

Binärer Bit-String B : $b_n \ b_{n-1} \ \dots \ b_3 \ b_2 \ b_1 \ b_0$

Gray Bit-String G : $g_n \ g_{n-1} \ \dots \ g_3 \ g_2 \ g_1 \ g_0$

$$g_j = \begin{cases} b_n & j = n \\ b_{j+1} \oplus b_j & j < n \end{cases} \quad b_j = \sum_{k=n}^j \oplus g_k$$



Eine optimale Kodierung wurde dann vorliegen, wenn eine kleine Änderung am Genotypen eine kleine Änderung am Phänotyp bewirkt und wenn eine grosse Änderung am Genotypen auch eine grosse Änderung am Phänotyp zur Folge hat.

8.4.2 Selektion Skript S. 15

Probabilistische (Zufallsprinzip, z.B. *Roulette-Wheel*) oder deterministische (nach Fitnesswerten, z.B. nur die besten; z.B. *Binäre Wettkampfselektion*) Selektion. Problem bei beiden: Die besten Gene können verloren gehen, deshalb werden meist das beste oder die besten Individuen übernommen.

8.4.3 Rekombination Skript S. 16

One-Point Crossover: Der „Chromosomen“-String wird an zufälliger Stelle getrennt und neu rekombiniert:

Elternteil 1: $1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0$

Elternteil 2: $0 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1$

Kind 1: $0 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0$

Kind 2: $1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1$

Two-Point Crossover: Der „Chromosomen“-String wird an zwei zufälligen Stellen getrennt und neu rekombiniert:

Elternteil 1: $1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0$

Elternteil 2: $0 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1$

Kind 1: $1 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0$

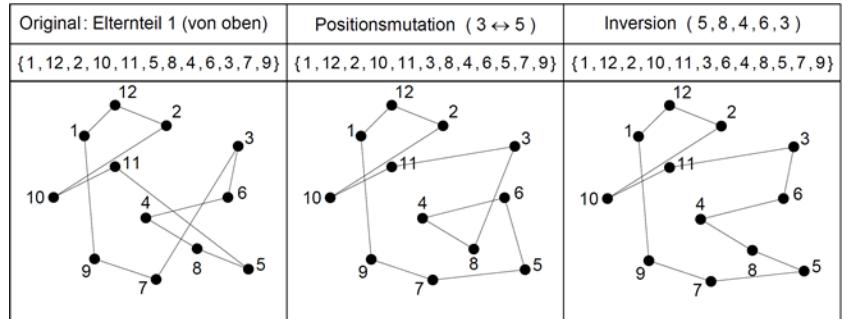
Kind 2: $0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1$

k-Point Crossover: Verallgemeinerung von One-/Two-Point Crossover.

Uniform Crossover: k -Point Crossover mit Grenzwert $k \rightarrow n$, hier wird also jedes Bit zufällig kombiniert.

8.4.4 Mutation Skript S. 18

- *Bit-Flip-Mutation*: Zufällig ausgewählte Bits werden invertiert, wobei dies mit (kleiner) Wahrscheinlichkeit (Mutationsrate) geschieht.
- *Positionsmutation*: Bits oder ganze Bitstrings können an verschiedenen Positionen ausgetauscht werden.
- *Inversion*: Bits können auch invertiert werden (Auswahl durch Zufall).



8.4.5 Ersetzungsschemata Skript S. 19

Was passiert mit der alten Population?

Ersetzungsschema	Vorteil	Nachteil	Ersetzungsschema	Vorteil	Nachteil
Generational Replacement Alle Individuen einer Generation werden durch ihre Nachkommen ersetzt	Die Dominanz einiger weniger Individuen kann durchbrochen werden	Die Bewertung der Nachfolger kann schlechter ausfallen als die der Vorgänger, d.h. die Fitness steigt nicht zwingend monoton	Delete-n-last Die schlechtesten n Individuen werden ersetzt. Ist n sehr klein, so sprechen wir vom steady-state Ersetzungsschema; das andere Extrem ist <i>Generational Replacement</i>	Monotonie der Fitness. Als Variante kann hier die Population von Generation zu Generation zunehmen	Gefahr eines lokalen Optimums, falls die Populationen gleich mächtig bleiben. Verfahren ist kompliziert, falls die Populationen zunehmen
Elitismus Die besten m Individuen werden direkt in die neue Generation übernommen	Die Fitness nimmt von Generation zu Generation monoton zu	Es besteht die Gefahr, dass aufgrund einer dominanten Elite der Optimierungsprozess zu früh in einem lokalen Optimum endet	Delete-n n zufällig ausgewählte Individuen werden ersetzt		Die Monotonie der Fitness ist nicht garantiert
Schwacher Elitismus Die Elite (die besten m Individuen) wird einer Mutation unterzogen, bevor sie weitere Nachkommen erzeugen kann	Der Nachteil des Elitismus, das Hängenbleiben in einem lokalen Optimum, wird reduziert.	Die Monotonie der Fitness ist nicht garantiert	Altersheim Ein Teil der Population wird für ein paar Generationen in einem „Altersheim“ aufbewahrt und dann wieder zur Fortpflanzung zugelassen	Der besten Partner für ein (gutes) Individuum könnte erst in einer späteren Generation folgen	Komplizierteres Verfahren

8.5 Ant Colony Optimization Skript S. 21

Ameisen folgen intensiven Pheromon-Spuren, da dort die Chance auf Futter zu treffen gross ist. Diese Idee wird bei ACO aufgenommen. Mit einer Wahrscheinlichkeit $P(x_{ij})$ wird ein möglicher Pfad ij in die neue Generation übernommen. Zusätzlich wird meist auch ein „Elite-Bypass“ mit Schwellwert $Q \in [0, 1]$ implementiert:

$$P(x_{ij}) = \begin{cases} \frac{\tau_{ij}^\alpha \eta_{ij}^\beta}{\sum_{j \in \mathbf{W}_i} \tau_{ij}^\alpha \eta_{ij}^\beta} & \text{if } j \in \mathbf{W}_i \\ 0 & \text{if } j \notin \mathbf{W}_i \end{cases}$$

$$P(x_{ij}) = \begin{cases} \text{if } z < Q = \begin{cases} 1 & \text{if } j = \arg \max_{k \in \mathbf{W}_i} [\tau_{ik}^\alpha \eta_{ik}^\beta] \\ 0 & \text{else} \end{cases} \\ \text{if } z \geq Q = \begin{cases} \frac{\tau_{ij}^\alpha \eta_{ij}^\beta}{\sum_{j \in \mathbf{W}_i} \tau_{ij}^\alpha \eta_{ij}^\beta} & \text{if } j \in \mathbf{W}_i \\ 0 & \text{else} \end{cases} \end{cases}$$

mit η_{ij} als heuristischer Information (z.B. Abstand $\eta_{ij} = \frac{1}{d_{ij}}$ beim TSP), z ist eine gleichverteilte Zufallsvariabale im Bereich $[0, 1]$.

Bei jedem Schritt verdunsten die Pheromone nach folgender Regel mit Verdunstungsfaktor $0 \leq \rho \leq 1$:

$$\tau_{ij}^{neu} = \begin{cases} \tau_{ij}^{alt}(1 - \rho) + \frac{\rho}{f(\bar{x})} & \text{if } j \in \mathbf{W}_i \\ \tau_{ij}^{alt}(1 - \rho) & \text{if } j \notin \mathbf{W}_i \end{cases} \quad \tau_{ij}^{neu} = \varphi \cdot \tau^0 + (1 - \varphi) \cdot \begin{cases} \tau_{ij}^{alt}(1 - \rho) + \frac{\rho}{f(\bar{x})} & \text{if } j \in \mathbf{W}_i \\ \tau_{ij}^{alt}(1 - \rho) & \text{if } j \notin \mathbf{W}_i \end{cases}$$

ACO kann lokalen Extremalstellen konvergieren. Die kann durch das einführen einer Konvergenzbremse φ abgeschwächt werden.

ACO wird eingesetzt für TSP, Vehicle Routing, Graph Coloring, Routing, Scheduling, etc.

ACO liefert schnell gute Resultate für kombinatorische Probleme, ist aber nicht für Parameteroptimierung geeignet.

```
/**Ant Colony Optimization*/
class ACO {
    static final int n      = 2;                      // Anzahl Design Variables
    static final int nbit = 20;                        // Anzahl Bits fuer Codierung
    static final int ngen = 150;                       // Anzahl Iterationen
    static final int npop = 200;                       // Populationsgroesse
    static final double xmin = -5.0;                  // x-range
    static final double xmax = 5.0;                   // 
    static double Delta = (xmax-xmin)/Math.pow(2.0, nbit/2);
    static double Offset = 0.0;                        // Offset for function weighting
    static double evap   = 0.2;                        // evaporation factor
    static double Q      = 0.05;                       // Schwellwert / Elite-Bypass
    static double phi    = 3.0;                         // Pheromon-Ausgleichs-Parameter (verhindert zu schnelle
    // objective function f (Rosenbrock function / Banana function)
    static double Obj_f(double[] x) {
        return (1.0-x[0])*(1.0-x[0]) + 100.0*(x[1]-x[0]*x[0])*(x[1]-x[0]*x[0]);
    }
}
```

```

// Decoding bits —> double
static double[] getVal(int[] bit) {
    double[] x = new double[n];           // Values
    double fac = 1.0;

    for (int m = 1; m<=nbit/n; m++) {   // Decoding bits —> double
        for (int j = 0; j<n; j++)
            x[j] = x[j] + fac*bit[(j+1)*nbit/n-m];
        fac= fac*2.0;
    }
    for (int j = 0; j<n; j++) x[j] = xmin+x[j]*Delta; // Scaling to grid
    return x;
}

public static void main(String[] args) {
    int [][] ant = new int [npop][nbit];      // Ants — coded design variables
    int [] bestAnt= new int [nbit];             // best Ant
    double[] x = new double[n];                 // Design Variablen
    double [][] tau = new double[nbit][n];       // Pheromone Matrix
    double [][] Dtau= new double[nbit][n];       //
    double [] tauTot= new double[nbit];          // Pheromone total pro moeglicher Wert
    double fbest=1000.0;                        // global best fitness
    double fnew;
    int pbest=0;                                // index best ant
                                                // Genesis
    for (int p = 0; p<npop; p++) {
        for (int b = 0; b<nbit; b++) ant[p][b] = (int) (2*Math.random());
        fnew = Obj_f(getVal(ant[p]));
        if (fnew < fbest) {
            pbest=p;                           // Index best ant
            fbest=fnew;
        }
        for (int b = 0; b<nbit; b++)
            tau[b][ant[p][b]] = tau[b][ant[p][b]] + evap /(Offset+fnew);
    }
    for (int b = 0; b<nbit; b++) bestAnt[b]= ant[pbest][b];
// hier starten die Ameisen
    for (int g=1; g<=ngen; g++) {              // Loop "Generationen" / Iterationen
        for (int p = 0; p<npop ; p++) {         // Loop ueber alle Ameisen
            for (int b = 0; b<nbit; b++) {
                tauTot[b]=tau[b][0];
                for (int m = 1; m<n; m++) tauTot[b] += tau[b][m];
            }
            if (Math.random() < Q )             // Schwellwert / Elite-Bypass
                for (int b = 0; b<nbit; b++) ant[p][b]= bestAnt[b];
            else {
                for (int b = 0; b<nbit; b++)
                    if( Math.random() < tau[b][1]/tauTot[b]) ant[p][b]= 1;
                    else ant[p][b]= 0;
            }
        }                                       // Pheromone update
        for (int b = 0; b<nbit; b++)
            for (int m = 0; m<n; m++) Dtau[b][m] = 0.0;
        fbest=1000.0;
        for (int p = 0; p<npop ; p++) {         // und finden beste Loesung
            fnew = Obj_f(getVal(ant[p]));
            if (fnew < fbest) {
                pbest=p;                         // Index best ant
                fbest=fnew;
            }
        }
    }
}

```

```

for (int b = 0; b<nbit; b++) // Pheromone Matrix auffuellen
    Dtau[b][ant[p][b]] = Dtau[b][ant[p][b]] + 1.0 /(Offset+fnew);
}
for (int b = 0; b<nbit; b++) // Pheromone update und Verwitterung
    for (int m = 0; m<n; m++) tau[b][m] = (1.0-evap)*tau[b][m] + evap * Dtau[b][m];
for (int b = 0; b<nbit; b++) bestAnt[b]= ant[pbest][b]; // neue beste Loesung
for (int b = 0; b<nbit; b++) { // Pheromon-Ausgleich
    tau[b][0] = (phi*tau[b][0] + tau[b][1])/(1.0+phi);
    tau[b][1] = (tau[b][0] + phi*tau[b][1])/(1.0+phi);
}
System.out.print("Iteration "+g);
x= getVal(bestAnt);
for (int m = 0; m<n; m++) InOut.print(x[m], 5,6);
System.out.println("f"+Obj_f(x));
}
}
}

```

8.6 Particle Swarm Optimization Skript S. 23

Speziell für Parameteroptimerung geeignet: Jedes Individuum versucht zwar, sein lokales Optimum zu finden, gleichzeitig lernt es aber von den seinen eigenen Erfahrungen und denen des Schwarms.

Particle Swarm Optimization : 1.) Wir wählen für jedes Individuum $j \in \{1, 2, 3, \dots, n\}$ (Partikel) im Schwarm eine Startnäherung $\vec{x}_j(0)$ für die Design-Variablen und eine Start-Geschwindigkeit $\vec{v}_j(0)$.

2.) Für jedes Individuum (Partikel) im Schwarm berechnen wir für die aktuelle Position $\vec{x}_j(t_i)$ die Fitness. Ist diese besser als die je schon vom Individuum erreichte Fitness, so setzen wir

$$\vec{x}_j^{(\text{best})} = \vec{x}_j(t_i).$$

Mit $\vec{x}^{(\text{global best})}$ bezeichnen wir die beste Position, welche bisher von irgend einem Partikel erreicht wurde.

3.) Für jedes Individuum $j \in \{1, 2, 3, \dots, n\}$ berechnen wir die neue Geschwindigkeit und die neue Position mit der Formel :

$$\vec{v}_j(t_{i+1}) = \underbrace{w \cdot \vec{v}_j(t_i)}_{\text{Trägheit}} + \underbrace{c_1 \cdot z_1 \cdot (\vec{x}_j^{(\text{best})} - \vec{x}_j(t_i))}_{\text{Individuelle Erinnerung}} + \underbrace{c_2 \cdot z_2 \cdot (\vec{x}^{(\text{global best})} - \vec{x}_j(t_i))}_{\text{Soziale Erfahrung}}$$

$$\vec{x}_j(t_{i+1}) = \vec{x}_j(t_i) + \vec{v}_j(t_{i+1})$$

4.) Schritte 2.) und 3.) wiederholen, bis ein vorgegebenes Abbruchkriterium erreicht ist. Der Zeitschritt hat dabei keine Bedeutung, t_{i+1} folgt in einer willkürlichen Einheit dem Zeitpunkt t_i .

Der Parameter w (*inertia weight*) beschreibt die Trägheit der Individuen. In der Regel ist dies eine Zahl kleiner 1, welche im Laufe der Zeit abnimmt. c_1 bezeichnen wir als kognitiven Parameter, er gewichtet die individuelle Erinnerung. c_2 ist ein sozialer Parameter, er gewichtet die Erfahrung des Schwarms resp. dessen besten Repräsentanten. z_1 und z_2 sind gleichverteilte Zufallszahl im Intervall $[0, 1]$. Weil diese Zufallszahlen im Mittel gleich 0.5 sind, werden die beiden Parameter c_1 und c_2 häufig $c_1 = c_2 = 2$ gesetzt.

```

/* Particle Swarm Optimization */

class PSO {

    static final int n      = 5;           // number of design variables
    static final int ngen = 150;          // number of generations
    static final int npop = 100;          // Population size
    static final double xmin = -5.0;       // range
    static final double xmax =  5.0;       //

    static double w = 1.0;             // inertia weight
    static double c1=2.0;            // cognitive parameter
    static double c2=2.0;            // social parameter

// objective function f (Rosenbrock function / Banana function)
    static double Obj_f(double[] x) {
        //return (1.0-x[0])*(1.0-x[0]) + 100.0*(x[1]-x[0]*x[0])*(x[1]-x[0]*x[0]);
        return 2.0*(x[0]*x[0] + x[1]*x[1] - 2.0*x[0] - 2.0*x[1] - x[2] - x[3])
            + x[2]*x[2] + x[3]*x[3] +(15.0+x[4]*x[4])/2.0 - x[4];
    }

    public static void main(String[] args) {
        double[][] x = new double[npop][n];           // design variable vector's
        double[][] v = new double[npop][n];           // design velocity vector's
        double[] f = new double[npop];                // best "Fitness"
        double[][] ibest= new double[npop][n];        // individual best position
        double[] gbest = new double[n];              // global best position
        double fbest;                                // global best fitness
        double fnew;                                 // Genesis

        for (int p = 0; p<npop; p++) {
            for (int m = 0; m<n; m++) x[p][m] = xmin+(xmax-xmin)*Math.random();
            f[p] = 10000.0;
        }
        fbest = 10000.0;

// hier startet die "Evolution"
        for (int g=1; g<=ngen; g++) {                  // loop over generations

            for (int p = 0; p<npop ; p++) {           // search individual best and global best
                fnew=Obj_f(x[p]);
                if ( fnew < f[p] ) {                  // individual best
                    f[p] = fnew;
                    for (int m = 0; m<n; m++) ibest[p][m]= x[p][m];
                    if ( fnew < fbest ) {            // global best
                        fbest = fnew;
                        for (int m = 0; m<n; m++) gbest[m]= x[p][m];
                    }
                }
            }

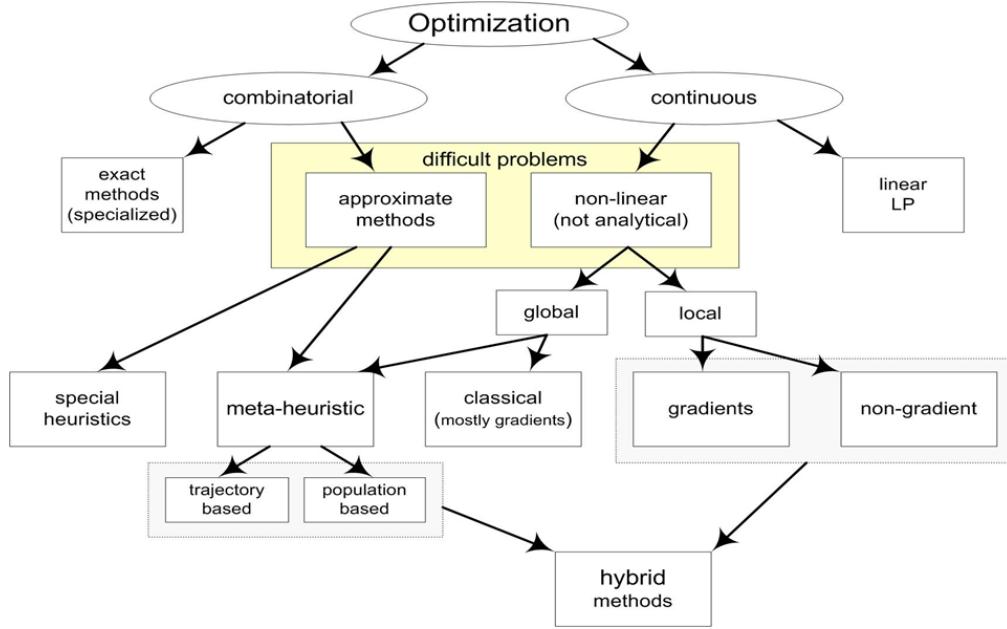
            System.out.print("generation "+g);
            for (int m = 0; m<n; m++) InOut.print(gbest[m], 5,6);
            System.out.println(" "+w+"+w+" "+f+" "+fbest);
            for (int p = 0; p<npop ; p++)           // update velocity & position
                for (int m = 0; m<n; m++) {
                    v[p][m] = w*v[p][m] + c1*Math.random()*(ibest[p][m] - x[p][m])
                                + c2*Math.random()*(gbest[m] - x[p][m]);
                    x[p][m] = x[p][m] + v[p][m];
                }
        }
    }
}

```

```
w=w*0.99;
}
}
}
```

// inertia weight changes

9 Optimization Summary



Auswahlhilfe für Algorithmen:

- Kontinuierliche Lineare Probleme: Lineare Programmierung, Integer Programmierung
- Kontinuierliche nichtlineare Probleme: Gradientenverfahren (Steepest Descent, Newton) oder bei komplexeren Problemen Heuristiken (Ant Colony oder Genetische Algorithmen)
- Einfache/gutartige Probleme: Trajektorienbasierte Algorithmen (Hill-Climbing, Tabu Search, Simulated Annealing)
- Komplexe Kombinatorische Probleme: Ant Colony oder Genetische Algorithmen
- Komplexe Parameteroptimierungen: Particle Swarm Optimization oder Genetische Algorithmen