Überblick über Optimierung

Modell

⇨ Ein Optimierungsproblem muss stets von der Realität abstrahieren

Modell: Präzise Formulierung von Objekten und deren Beziehungen (so detailliert wie nötig und so einfach wie möglich)

Optimierungsproblem:

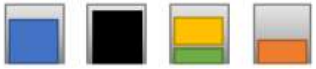
- Entscheidungsvariablen
- Nebenbedingungen mit allen Parametern
- Zielfunktion

⇨ Beispiel mit TSP:
Entscheidungsvariablen: «Welche Stadt ist wann an der Reihe?»
Nebenbedingungen: «Jede Stadt genau einmal, genau ein Tour»
Zielfunktion: Länge der Tour minimieren

Beispiel: Bin Packing Problem

Bin Packing Problem

Gegeben: n Gegenstände mit der Grösse $a_1, \ldots, a_n \in (0, 1]$.
Gesucht: eine Zuordnung in Behälter (Bins) $B_1, B_2, \ldots, B_m$ sodass $\sum_{a_i \in B_k} a_i \leq 1$ für jeden Behälter $k = 1, \ldots, m$ gilt, sodass die Anzahl genutzter Behälter m minimiert wird.



Enumerativer Algorithmus: (gestartet mit tiefensuche(1))

function *tiefensuche*(Gegenstand i)

- Falls $i > n$: Aktuelle Lösung ausgeben.
- Falls $i \leq n$: Für Behälter B_k mit $k = 1, \ldots, n$:
  - Falls Gegenstand i in Behälter B_k passt:
    - Gegenstand i in Behälter B_k packen.
    - Rufe *tiefensuche*$(i + 1)$ auf.
    - Gegenstand i aus Behälter B_k entfernen.

⇨ Zuordnungen vermeiden, in denen Behälter mit kleinerem Index leer bleiben? **Stoppe falls Bk nun leer ist**

⇨ Bei n Gegenständen n! Funktionsaufrufe

⇨ Tiefensuche beschleunigen, indem man bei jedem Rekursions-Knoten abschätzt ob er zu einer besseren Lösung führen könnte (Branch & Bound)

Bin Packing Problem: Abschätzung auf eine einfache Weise

- Ein Knoten beschreibt mit Teillösung x' die Behälterzuordnung für einen Teil der Gegenstände.
- Ein Vergleich der aktuellen Anzahl verwendeter Behälter (hier bezeichnet als *bound*(x')) mit der bisher kleinsten Anzahl $f^\triangle$.
- Der Knoten wird also übergangen, wenn *bound*$(x') < f^\triangle$.

global

⇨ Einfach gesagt: Wenn aktuelle Anzahl an Behälter bereits grösser ist als bisher gefundene kleinste Anzahl für eine Lösung dann brich in diesem Teilbaum ab.

Branch&Bound Tiefensuche

1. Setze $f^\triangle$ auf einen ungültigen Wert. Optional: durch Heuristik eine bisher beste Lösung $x^\triangle$ bestimmen und $f^\triangle \leftarrow f(x^\triangle)$ setzen.
2. Sei $x \leftarrow \emptyset$ eine leere Teillösung.
3. Rufe *b&b-tiefensuche*(x) auf.
4. Optimale Lösung $x^* \leftarrow x^\triangle$ ausgeben.

function *b&b-tiefensuche*((Teil-)Lösung x)

- Falls x eine vollständige Lösung ist:
  - Falls $f(x)$ besser ist als $f^\triangle$, dann $x^\triangle \leftarrow x$ und $f^\triangle \leftarrow f(x^\triangle)$ setzen.
- Sonst: Für jede (Teil-)Lösung $x' \in$ *branches*(x):
  - Falls *bound*(x') besser ist als $f^\triangle$:
    - Rufe *b&b-tiefensuche*(x') auf.

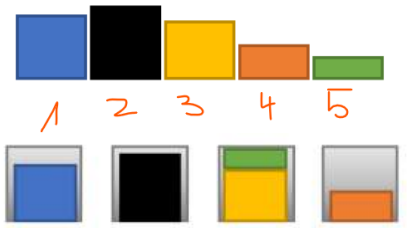
- Das Branching passiert mit dem if bei Falls bound …, denn wenn schlechter oder gleich dann versuche gar nicht erst weiter zu gehen.
- f(x) ist der Zielfunktionswert (hier einfach die Anzahl der Bins von x) und x die jeweilige Teillösung (bzw. volle Lösung wenn Blatt erreicht). x kann je nach Format bspw. auch Vektor sein mit 1 und 0 und 1 wenn an index i (Bin i) enthalten sonst 0. f(x) addiert dann einfach alle Komponenten auf.
- Heuristik: Vereinfachte Entscheidungs- oder Berechnungsmethode, die eingesetzt wird, um in komplexen oder rechenaufwändigen Problemen schnell zu einer genug guten Lösung oder Einschätzung zu kommen (keine Garantie für optimale Lösung).

First Fit Heuristik:

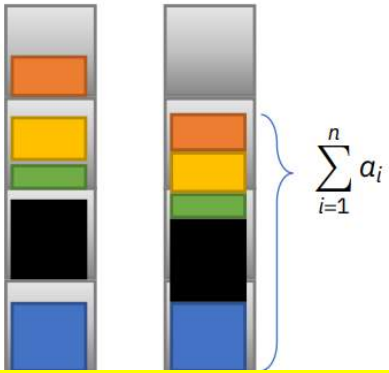
Gegenstände werden in zufälliger Reihenfolge betrachtet.
Behälter bezeichnen wir mit $B_1, B_2, \ldots, B_m$.
Zu Beginn gibt es $m \leftarrow 0$ Behälter.

- Wiederhole für jeden Gegenstand $i = 1, \ldots, n$:
  - Falls möglich, lege i in den ersten Behälter B_k, $k \in \{1, \ldots, m\}$, in welchem noch genügend Platz frei ist.
  - Sonst: lege i in einen neuen Behälter B_{m+1} und setze $m \leftarrow m + 1$.

⇨ Beispiel:



- ⇨ Untere Schranken Berechnung für die Anzahl an benötigter Behälter, die mindestens verwendet werden müssen:



- ⇨ In bound(x') wie folgt verwendbar: bound(x') = Anzahl bereits benutzter Bins + untere Schranke (der Elemente noch nicht in der Lösung / zu Bins zugewiesen) => Wenn bound(x') >= f delta dann Pruning (abschneiden)
- ⇨ Am Ende sind min. m-1 Bins halbvoll: Wenn 2 Bins <= 0.5 haben könnte man sie umfüllen (algo kann solchen Fall nicht erzeugen)
- ⇨ Höchstens doppelt so viele wie min nötig

$$\frac{m-1}{2} < \sum_{i=1}^{n} a_i \leq m^* \text{ bzw. } m - 1 < 2m^*$$

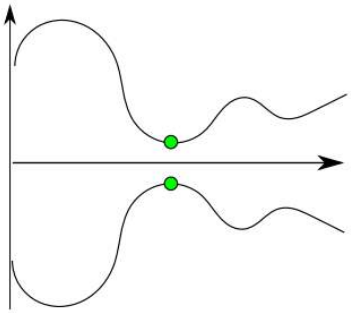
Begriffe

Problem vs. Problem-Instanz:

- Optimierungsproblem: Allgemeine Definition einer Fragestellung (Parameter als Platzhalter etc.)
- Instanz eines Problems: Konkrete Ausgestaltung des Problems mit Werten für alle Parameter

Minimierung vs. Maximierung:

$$\min\{f(x) \mid x \in S\} = -\max\{-f(x) \mid x \in S\}$$



Optimum:

- Optimale Lösung x erreicht optimalen Zielfunktionswert
- Optimierungsproblem kann mehrere optimale Lösungen besitzen

Nachbarschaft:

- Eine mengenwertige Funktion, welche jedem Punkt (in einer Menge M) eine Menge von benachbarten Punkten (die eine Teilmenge von M sind) zuordnen Im TSP bspw. Menge von allen Touren

Lokale vs. globale Optima:

- Lokales Optimum: Lösung x, die im vergleich zu allen Lösungen in ihrer Nachbarschaft den besten Zielfunktionswert erreicht
- Globales Optimum ist stets auch ein lokales Optimum

Nachbarschaftssuche:

- Iterativ Lösung x mit einer besseren Lösung x' aus ihrer Nachbarschaft austauschen, bis x lokal optimal ist
- Hier ausschliesslich kombinatorische Optimierung: Eine Lösung x ist dann ein diskreter Vektor
- Optimale (globale und lokale) Lösungen sind oft schwer zu finden (NP-Vollständig). Sogar polynomiell lösbare Probleme wie kürzester Weg können zu aufwändig sein.

Enumerativer Algorithmus:

- Verfahren, das *alle* zulässigen Lösungen auflistet.

Exakter Algorithmus:

- Verfahren, das eine optimale Lösung findet
- Im Gegensatz zu Heuristische Lösung: Erlaubt eine Abweichung zur optimalen Lösung. Das ermöglicht einen schnelleren Algorithmus

Heuristik vs. Metaheuristik:

- Heuristik: Verfahren, das eine Lösung sucht, für die weder die Zulässigkeit noch die Optimalität garantiert ist.
- Heuristik: Problemspezifischer Algo
- Metaheuristik: Wendet gleiche Grundprinzipen auf verschiedene Probleme an

Nachbarschaft TSP Beispiel: Menge M ist die Menge aller möglichen Touren und ein Punkt darin (also eine Tour) dessen Nachbarschaft ist eine Teilmenge von M (also eine Menge von Touren). Und zwar die, von denen bspw. durch 2-OPT der aktuellen Tour entstehen können

Metaheuristiken: Konstruktive Algorithmen

Definition

Baut eine Lösung Stück für Stück auf, bis sie vollständig ist.

- ⇨ Beispiel: Greedy Verfahren
- ⇨ Konkrete Anwendungsbeispiele: Kürzester Pfad von A nach B, minimalen Spannbaum eines Graphens
- ⇨ Auch heuristische Lösungen sind machbar (Bin Packing, TSP, Knotenfärbung)

Greedy Verfahren

- Konstruiert sukzessiv eine optimale Lösung in Bezug auf Zielfunktion f (greedy = gierig = stets das beste Stück wird genommen)
- Gewichtsfunktion w misst die Güte einer Teillösung
- Es soll eine Lösung mit max w-Wert konstruiert werden
- Exakte Greedy Verfahren eignen sich dazu optimale Lösungen zu bestimmen (besonders wenn Matroid-Eigenschaft erfüllt wird) => Andernfalls handelt es sich um eine Greedy Heurisitk
- Greedy Approximation: Teilweise lässt sich Approximationsgüte zeigen / beweisen (bspw. First Fit in Bin Packing)

Matroid:

Sei E eine endliche Menge und sei $\mathcal{U}$ eine Menge von Teilmengen von E.

Teilmengensystem

Die algebraische Struktur $(E, \mathcal{U})$ heisst ein *Teilmengensystem*, falls gilt:

1. $\emptyset \in \mathcal{U}$.
2. Wenn $A \subseteq B$ und $B \in \mathcal{U}$ dann ist $A \in \mathcal{U}$.

Ist folgendes $(E, \mathcal{U})$ ein Teilmengensystem?

$E = \{a, b, c\}, \quad \mathcal{U} = \{\emptyset, \{a\}, \{b\}, \{c\}, \{b, c\}\}$.

1 ✓ 2 ✓

- ⇨ Teilmengensystem wenn leere Menge in U und für jedes Element in U (was auch eine Menge ist) müssen alle ihre Teilmengen in U sein.
- ⇨ **E ist eine Menge und U hat Teilmengen von E**

Matroid

Ein Teilmengensystem $(E, \mathcal{U})$ heisst *Matroid*, falls die *Austauscheigenschaft* gilt:
Für eine Menge $A \in \mathcal{U}$ die weniger Elemente hat als eine Menge $B \in \mathcal{U}$ muss es ein Element $x \in B \setminus A$ geben sodass $(A \cup \{x\}) \in \mathcal{U}$.

Gegenbeispiel: Zeigen Sie, dass folgendes Teilmengensystem $(E, \mathcal{U})$ die Austauscheigenschaft *nicht* erfüllt und daher kein Matroid ist:

$E = \{a, b, c\}, \quad \mathcal{U} = \{\emptyset, \{a\}, \{b\}, \{c\}, \{b, c\}\}$.

A = {a}, B = {b, c}.

$B \setminus A = \{b, c\} = X$

$\{a, b\} \notin \mathcal{U}$

- ⇨ Achtung: Mindestens ein Element x in B \ A sodass A vereinigt {x} in U

Kanonischer (allgemeiner) Greedy Algo:

- Gegeben Teilmengensystem $(E, \mathcal{U})$, Gewichtsfunktion $w : E \to \mathbb{R}$.
- Wir suchen in $\mathcal{U}$ eine (bzgl. $\subseteq$) maximale Menge T, sodass $f(T) = \sum_{e \in T} w(e)$ minimiert wird.

Kanonischer Greedy Algorithmus

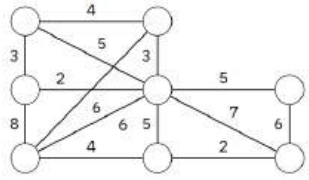
- Ordne die Elemente in E nach aufsteigendem Gewicht sodass $w(e_1) \leq \cdots \leq w(e_n)$.
- Setze $T \leftarrow \emptyset$.
- Für $k = 1, \ldots, n$:
  - Wenn $T \cup \{e_k\} \in \mathcal{U}$, dann setze $T \leftarrow T \cup \{e_k\}$
- Gib die Lösung T aus.

- ⇨ Wenn (E, U) Matroid, dann Algorithmus optimal (bzw. exakt)
- ⇨ Soll f(T) maximiert werden, dann Elemente absteigend sortieren

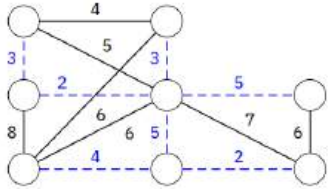
Problem: Minimaler Spannbaum

Gegeben ist ein gewichteter Graph $G = (V, E, w)$ mit:

- Knotenmenge V,
- Kantenmenge $E \subseteq V \times V$,
- Gewichtsfunktion $w : E \to \mathbb{R}$.



- ⇨ Wir suchen min. Spannbaum (ein Baum der alle Knoten miteinander verbindet, wobei Summe aller Kantengewichte minimal ist)



Das Matroid sorgt nur dafür, dass du **gültige Mengen aufbauen kannst**.
Der Greedy-Algorithmus sorgt dann dafür, dass du **lokal beste Entscheidungen triffst** – auch mit negativen Werten.

- ⇨ Lösbar mit kanonischem Greedy
- ⇨ E sei die Menge aller Kanten, jedes T in U sei eine zyklenfreie Teilmenge von Kanten in E und w sei das Kantengewicht
- ⇨ Deshalb Matroid: Keine Zyklen (also Unabhängigkeit erhalten)
- ⇨ Austauscheigenschaft versichert, dass alle maximalen unabhängigen Mengen gleich gross sind (wie Rang in der linearen Algebra) und dass man eine kleinere unabhängige Menge schrittweises Ergänzen zur Grösse einer grösseren erweitern kann (ohne Unabhängigkeit zu verletzen).
- ⇨ Warum unabhängig? Weil nur unabhängige Mengen zulässige Teillösungen sind (bspw. sonst Zyklen)

$I = \{(A,B),(B,C)\}$ → unabhängig

$I = \{(A,B),(B,C),(C,A)\}$ → **nicht** unabhängig (weil Kreis enthalten)

Algo für Greedy min. Spanning Tree:

1. Zuerst: alle Kanten nach Gewicht sortieren.
2. Dann Kanten sukzessive zur Teillösung T hinzufügen (sofern dabei kein Zyklus entsteht).

- ⇨ Kante (a,b) darf nicht hinzugefügt werden, wenn zwischen a und b bereits ein Pfad existiert (bspw. mittels DFS)

Problem: Kürzester Pfad

- ⇨ Gleicher Aufbau wie Spanningtree, jedoch mit Startknoten und wir suchen kürzesten Pfad zu jedem Knoten

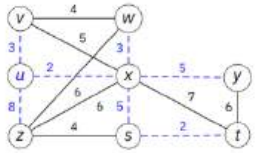
- Der Kanonische Greedy Algorithmus findet stets, ausgehend von Startknoten $u \in V$, den kürzesten Pfad zu *jedem* Knoten in V.
- Beweist die Korrektheit des Algorithmus von Dijkstra (1959).
- Wir nutzen hierfür das Teilmengensystem $(P, \mathcal{U})$ und die Gewichtsfunktion $w' : P \to \mathbb{R}$.
  - P sei die Menge *aller* zyklenfreier Pfade vom Startknoten u aus,
  - jedes $T \in \mathcal{U}$ besteht aus allen solchen Pfadmengen, die auf *verschiedene* Endknoten führen,
  - $w'(p) = \sum_{k \text{ liegt auf } p} w(k)$ beschreibt die Länge eines Pfades $p \in P$.
- Für die Optimalität ist wieder zu zeigen, dass $(E, \mathcal{U})$ ein Matroid ist:
  - In $A \in \mathcal{U}$ gibt es $|A|$ verschiedene Endknoten.
  - In $B \in \mathcal{U}$ mit $|B| > |A|$ befindet sich mindestens ein Pfad mit einem weiteren Endknoten, der in A nicht vorkommt.
  - Daher kann A um diesen Pfad erweitert werden.

⇨ Vorallem letzter Punkt wichtig, welcher den Grund für einen Matroid zeigt (man kann A erweitern, so dass kein Zyklus entsteht).

1. Zuerst: Alle zyklenfreie Pfade von u aus ermitteln und diese nach Länge w' aufsteigend sortieren. $O((n-1)!) \cdot O(n \log n)$
2. Dann Pfade sukzessive zur Teillösung T hinzufügen (sofern der Endknoten des Pfads bisher noch nicht erreicht ist. So gibt es dann je Endknoten einen Pfad). $O((n-1)!)$

Beispiel:

Pfad $p \in T$	$w'(p)$
u	0
$u-x$	2
$u-v$	3
$u-x-w$	5
$u-x-y$	7
$u-x-s$	7
$u-z$	8
$u-x-s-t$	9



- ⇨ Achtung: Alle Endknoten verschieden
- ⇨ Verfahren ist ineffizient (da zuerst alle Pfade erzeugt werden müssen)
- ⇨ Deshalb Dijkstra:

W	Liste von noch zu sondierenden Knoten (zu Beginn ist $W = V$ und am Ende ist $W = \emptyset$)
F	Auswahl an Kanten, welche die kürzesten Pfade von u aus zu allen anderen Knoten ausmachen (Ist am Ende die Lösung)
$l(a)$	Aktuell kürzestmögliche Pfadlänge von u nach $a \in V$
$s(a)$	Optimal zu $a \in V$ führender Knoten $s(a) \in V \setminus W$

Algorithmus von Dijkstra (1959)

1. Für jeden Knoten $a \in V$ setze $l(a) \leftarrow \infty$
2. $W \leftarrow V$, $F \leftarrow \emptyset$, $l(u) \leftarrow 0$
3. Für $i = 1, \ldots, |V|$:
   1. Finde einen Knoten $a \in W$ mit $l(a)$ minimal
   2. Setze $W \leftarrow W \setminus \{a\}$
   3. Wenn $a \neq u$, setze $F \leftarrow F \cup \{(k(a), a)\}$
   4. Für jeden Nachbarn b von a der auch in W enthalten ist und für den $l(b) > l(a) + w(\{a,b\})$ ist, setze $l(b) \leftarrow l(a) + w(\{a,b\})$ und $k(b) \leftarrow a$.

- Ist schneller, da Pfad zusammen mit Lösung aufgebaut wird
- Suche nach kleinstem l(a) am besten über Priorityqueue in O(log|W|)
- Wenn es nur um einen Zielknoten geht, kann der A* Algorithmus noch schneller sein:

W	Liste von noch zu sondierenden Knoten
F	Auswahl an Kanten, welche die kürzesten Pfade von Startknoten u aus zu allen anderen Knoten ausmachen
$l(a)$	Aktuell kürzestmögliche Pfadlänge von u zu einem Knoten a
$h(a)$	Monotone heuristische Mindest-Pfadlänge von a zum Zielknoten z
$s(a)$	Optimal zu $a \in V$ führender Knoten $s(a) \in V \setminus W$

A* Algorithmus (Hart&Nilsson&Raphael, 1968)

1. Für jeden Knoten $a \in V$ setze $l(a) \leftarrow \infty$
2. $W \leftarrow V$, $F \leftarrow \emptyset$, $l(u) \leftarrow 0$
3. Für $i = 1, \ldots, |V|$:
   1. Finde einen Knoten $a \in W$ mit $l(a) + h(a)$ minimal
   2. Setze $W \leftarrow W \setminus \{a\}$
   3. Wenn $a \neq u$, setze $F \leftarrow F \cup \{(k(a), a)\}$
   4. Für jeden Nachbarn b von a der auch in W enthalten ist und für den $l(b) > l(a) + w(\{a,b\})$ ist, setze $l(b) \leftarrow l(a) + w(\{a,b\})$ und $k(b) \leftarrow a$.

Problem: Knotenfärbung mit wenig Farben

- Gegeben: Graph (V, E) (V = {v1, …, vn}, E = VxV = {(v1, v1), (v1,v2),…,(vn, vn)})
- Gesucht: Eine Färbung c: V -> N, die jedem Knoten eine Farbe (als natürliche Zahl) zuordnet
- Benachbarte Knoten müssen verschiedene Farben haben
- Ziel: Minimierung der Anzahl genutzter Farben = «chromatische Zahl»
- NP-Vollständig diese Zahl zu bestimmen

Greedy Heuristik für die Knotenfärbung

1. Nummeriere die Knoten nach absteigenden Knotengraden:

$$d_1 \geq d_2 \geq \cdots \geq d_n$$

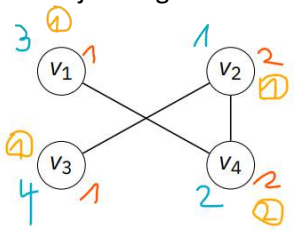
Dabei sei d_i der Knotengrad von Knoten $v_i \in V$.

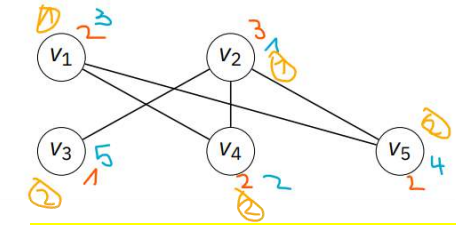
2. Setze $c(v_1) \leftarrow 1$.
3. Für $i = 2, \ldots, n$:

$$c(v_i) \leftarrow \min_{k \in \mathbb{N}} \{k \neq c(v_u) \mid u \in \{1, \ldots, i-1\}, (v_i, v_u) \in E\}$$

(kleinste Farbe zuordnen, die noch kein Nachbarknoten hat)

- ⇨ Knotengrad = Anzahl Nachbarn des jeweiligen Knotens





- ⇨ Rot: Grad, Blau: Reihenfolge (Nr. nach Grad), Gelb: Farbe
- ⇨ Da bspw. bei gleichem Grad unterschiedliche Reihenfolge gewählt werden kann, auch nicht optimale Lösung möglich

Knotenfärbung zum Sudoku-Lösen

Je Zeile/Spalte/Block soll jede Ziffer 1 ..., 9 genau einmal vorkommen.

- Ein Knoten je Kästchen
- Kante zwischen Kästchen in Zeile/Spalte/Block

	2		5		1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Gegeben

4	2	6	5	7	1	3	9	8
8	5	7	2	9	3	1	4	6
1	3	9	4	6	8	2	7	5
9	7	1	3	8	5	6	2	4
5	4	3	7	2	6	8	1	9
6	8	2	1	4	9	7	5	3
7	9	4	6	3	2	5	8	1
2	6	5	8	1	4	9	3	7
3	1	8	9	5	7	4	6	2

Gesucht

Problem: Traveling Salesmen (TSP)

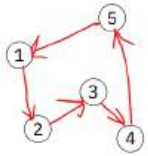
Ziel: Kürzeste Route finden (in einem vollständigen gewichteten Graph), um alle Städte (Knoten) genau einmal zu durchlaufen. Wir suchen Rundreise (Permutation bzw. PI)

- ⇨ NP-Vollständig
- ⇨ Es folgen heuristiken für nicht optimale Lösungen
- ⇨ => Bruteforce wäre optimal und aber O(n!)
- ⇨ Hier wird überall w = Metrisch (w(a,b) <= w(a,c) + w(c,a)) (Direkter Weg nie teurer als ein Umweg)

Nearest Neighbor Heuristik:

Nearest Neighbor Heuristik

Ausgehend von der Tour $\pi = (1)$ wird iterativ zum letzten Knoten $\pi(|\pi|)$ der nächste, noch nicht besuchte Nachbar gesucht und an π angehängt.



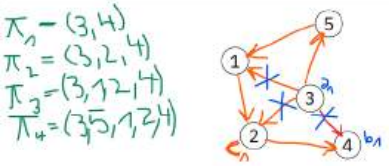
- ⇨ Kommt sehr auf Startknoten drauf an.
- ⇨ Man kann diesen n mal wiederholen (mit unterschiedlichen Startknoten und wählt günstigsten
- ⇨ Nicht Optimal weil lokal gesucht wird (die beste Route könnte auch zuerst eine gross Gewichtung haben die aber in diesem Algorithmus nie gewählt wird).

Cheapest Insertion Heuristik:

- ⇨ Nimmt nicht wie bei NN nächsten Knoten sondern fügt immer günstigsten Punkt ein, der am wenigsten die Gesamtkosten erhöht

Cheapest Insertion Heuristik

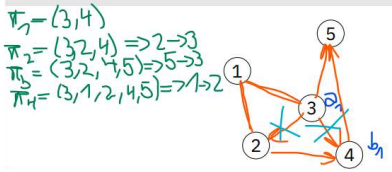
1. Beginne eine Tour $\pi = (a, b)$ wobei die Kante zwischen a und b die minimale Länge hat.
2. Für $i = 1, \ldots, n-2$:
   1. Selektiere denjenigen noch nicht besuchten Knoten c, der einen minimalen Wert $w(a,c) + w(c,b) - w(a,b)$ für ein in der Tour benachbartes Paar von Knoten a, b erreicht, also sei $c = \arg\min_{c' \notin \pi}(\min_{a,b \text{ benachbart in } \pi}(w(a,c') + w(c',b) - w(a,b)))$.
   2. Füge Knoten c in Tour π zwischen a und b ein.



Nearest Insertion Heuristik:

Nearest Insertion Heuristik

1. Beginne eine Tour $\pi = (a, b)$ wobei die Kante zwischen a und b die minimale Länge hat.
2. Für $i = 1, \ldots, n-2$:
   1. Selektiere denjeinigen noch nicht besuchten Knoten c, der minimalen Abstand zu einem Knoten v in der Tour hat, also sei $c = \arg\min_{c' \notin \pi}(\min_{v \in \pi}(w(c', v)))$.
   2. Füge Knoten c in π zwischen denjenigen in der Tour benachbarten Knoten a, b ein, bei denen $w(a,c) + w(c,b) - w(a,b)$ minimal ist.



- ⇨ Immer einen Knoten c wählen der min Abstand zu einem Knoten aus der Tour hat und dann dort einfügen, wo Zunahme der Gesamtkosten lokal minimal wird (Zunahme = Neu – Alt)

- ⇨ Finde Insertion Platz dort wo c kleinsten Zuwachs an Gesamtlänge bringt und wähle c dort wo Länge zu einem Tourknoten minimal

Farthest Insertion Heuristik:

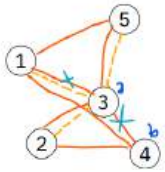
- ⇨ Wir suchen grössten minimalen Abstand zu Tourknoten

Farthest Insertion Heuristik

1. Beginne eine Tour $\pi = (a, b)$ wobei die Kante zwischen a und b die minimale Länge hat.
2. Für $i = 1, \ldots, n-2$:
   1. Selektiere denjenigen noch nicht besuchten Knoten c, der maximalen Abstand zu dem ihm nächstliegenden Knoten in der Tour hat, also sei $c = \arg\max_{c' \notin \pi}(\min_{v \in \pi}(w(c', v)))$.
   2. Füge Knoten c in π zwischen denjenigen in der Tour benachbarten Knoten a, b ein, bei denen $w(a, c) + w(c, b) - w(a, b)$ minimal ist.

$\pi_1 = (3,4)$
$\pi_2 = (3,1,4)$
$\pi_3 = (3,5,1,4)$
$\pi_4 = (3,5,1,4,2)$

max (min(...))
= max {5,1,2}
= 1



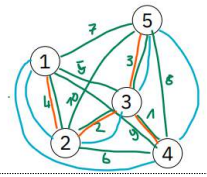
=> Achtung: Keine Kanten «entfernen» da Rundreise durchgehend erhalten (da PI = Permutation)
=> Auch hier wieder kleinste Zunahme beim Einfügen durch w(a, c) + w(c, b) – w(a, b)

Minimaler-Spannbaum Heuristik:

Minimaler-Spannbaum Heuristik

1. Berechne einen minimalen Spannbaum, Rechenzeit $O(|E| \log |V|) = O\left(\frac{n(n-1)}{2} \log n\right) = O\left(n^2 \log n\right)$
2. Durchlaufe in einer Tiefensuche den Baum beginnend in Knoten 1 und füge die dabei besuchten Knoten nacheinander in eine Tour ein, Rechenzeit $O(n)$

1. sort, add
2. $\pi = (1,2,3,5,4)$



- ⇨ Wegen Tree Traversal => O(n) (auch bei DFS)

Approximationsfaktoren:

Heuristik	α
Min. Spannbaum	2
Nearest Neighbor	$(\lceil \log_2 n \rceil + 1)/2$
Cheapest Insertion	2
Nearest Insertion	2
Farthest Insertion	$\log_2 n + 1$

Im metrischen TSP gibt es teils einen Approximationsfaktor α, d.h. für eine beliebige Instanz findet die Heuristik eine Lösung x mit $f(x) \leq \alpha \cdot f(x^*)$, dabei sei x^* sei eine optimale Lösung.

- ⇨ Wie viel schlechter die Heuristik im schlimmsten Fall zum Optimum sein kann
- ⇨ NN Faktor durch Clustern hergeleitet (innerhalb von Clustern Längen sehr kurz aber zwischen Clustern lang) der NN verbringt dabei die meiste Zeit innerhalb eines Clusters und geht zum nächsten wenn fertig und Optimale Tour springt zwischen Clustern hin und her. Binärbaum aus Clustern

2 TSP Arten:

Begriff	Bedeutung
Metrisches TSP	Jeder Graph erfüllt: 1. **Dreiecksungleichung**: $w(a,b) \leq w(a,c) + w(c,b)$ 2. **Symmetrie** (meist): $w(a,b) = w(b,a)$
Euklidisches TSP	Spezialfall vom metrischen TSP, bei dem die Knoten Punkte in $\mathbb{R}^2$ oder $\mathbb{R}^d$ sind und die Gewichte echte **euklidische Distanzen** sind: $w(a,b) = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}$

=> bei beiden min. Insertion $w(a,c) + w(c,b) - w(a,b)$ anwendbar

Verbesserungs-Algorithmen

Lokale Suche

Idee von Verbesserungs-Algorithmen:

- Konstruktive Heuristik Algorithmen erzeugen erste Lösungen
- Verbesserungs-Algorithmen verbessert diese

Lokale Suche:

1. Konstruiere eine Startlösung $x \in S$.
2. Wiederhole, solange es ein $x' \in N(x)$ mit $f(x') < f(x)$ gibt:[1]
   1. Wähle durch eine geeignete Methode ein $x' \in N(x)$ mit $f(x') < f(x)$.
   2. Setze $x \leftarrow x'$.
3. Lösung x ausgeben.

- Idee: Lösung x iterativ mit einer Lösung in ihrer Nachbarschaft N(x) tauschen, solange wie sich Zielfunktion auch verbessert.
- Fall nicht mehr verbessert wird, ist x bzgl. N(x) **lokal** optimal.
- Als Nachbar kann alles gelten (ist eine Funktion) also muss nicht unmittelbar gleich neben x liegen sondern einfach N(x) erfüllen.
- Im Algorithmus oben heisst «verbessert» = Zielfunktion minimieren

Methoden für Wahl von Nachbarlösung x':

Die übliche Wahl ist in Richtung der stärksten Verbesserung:
- *Best Improvement:* jene benachbarte Lösung, die zu einer maximalen Verbesserung führt
  - Aufwändig bei grosser Nachbarschaft, dafür ist die Schrittweite so am grössten.
- *First Improvement:* zur erstbesten, die zu einer Verbesserung führt; Betrachtet die Nachbarn in einer festgelegten Reihenfolge.
- *Least Improvement:* jene benachbarte Lösung, die die kleinste Verbesserung bewirkt

Auch eine nichtdeterministische Wahl ist möglich:
- *Uniform Random:* eine zufällig und gleichverteilt gewählte benachbarte Lösung
  - Eher wie eine zufällige Suche bei grosser Nachbarschaft.
- *Tournament:* beste Lösung einer zufälligen Teilmenge aus $N(x)$

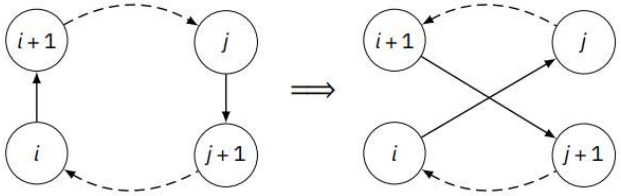
In diesem Zusammenhang ist je nach Grösse der Nachbarschaft die Zahl Iterationen und die Rechenzeit je Iterationsschritt abzuwägen.

ACHTUNG: Nachbarschaft ändert sich mit der aktuellen Lösung, bspw. ist 2-OPT-Nachbarschaft die Menge der zulässigen Lösungen, die in einem einzigen Schritt der aktuellen Lösung erreichbar sind

TSP 2-OPT

- 2-OPT Nachbarschaft N(x) hat eine Lösung für jede Knotenkombination mit i, j wobei i+1 < j => Dabei wird die Tour jeweils geändert (2-OPT da 2 Knoten jeweils geändert werden): Teil Tour wir umgekehrt

i+1<j



⇨ Eine Lösung (eine geänderte Tour in 2-OPT) ist dann eine Verbesserung wenn:

alt

$$w(i,i+1)+w(j,j+1) > w(i,j)+w(i+1,j+1)$$

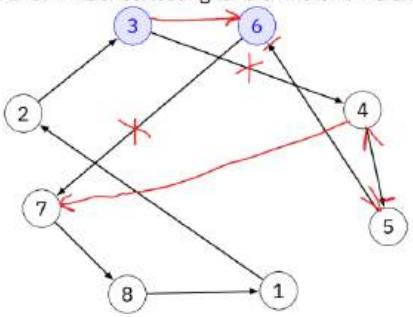
neu

- Zielfunktionswert Veränderung: **(d(vi,vj) + d(vi+1,vj+1)) – (d(vi,vi+1) + d(vj,vj+1)) => Wenn < 0 dann verbessert**
- Anzahl Möglichkeiten für die Wahl i, j: **Kleiner Gauss => ((n-1)(n-2))/2** (oder binomialkoeffizient mit 2 aus n-1) / 2 weil i + 1 < j => i + 2 <= j

Beispiel:

⇨ Evtl. Reihenfolge umkehren:

Wie ist die 2-OPT-Nachbarlösung für die Knoten $i = 3$ und $j = 6$.



TSP k-OPT

⇨ Erweiterung von 2-OPT auf k Kanten

⇨ k Kanten werden entfernt und die frei gewordenen Knoten so neu verbunden, dass wieder eine Tour entsteht (viele neue Verbindungsmöglichkeiten)

⇨ Anzahl der Verbindungsmöglichkeiten wächst exponentiell mit k => 2-OPT bis zu 2, 3-OPT bis zu 8 – 1 (7 neue)
(2^(k-1) * (k-1)!)

- Was geschieht wenn man ein grösseres k wählt?
  - Nachteil: Grosse Nachbarschaft ist aufwändiger zu durchsuchen
  - Vorteil: kann in der Lokalen Suche schneller besser werden und ein besseres lokales Optimum ergeben.

⇨ Alle ausgehenden Kanten der k Knoten entfernen und neu verbinden, dass wieder eine Tour entsteht

⇨ Implementierung:
Gegeben: Tour, Distanzmatrix (Entfernung zwischen Knoten i,j), Nachbarschaft auswählen (3 Kanten)
Grundidee: Entferne die gewählten Kanten, Versuche 3 Segment zu verbinden: nehme einen Knoten aus {i,j,k} und verbinde diese mit einem Punkt der nur 1 Kante hat, der nächste aus {i,j,k} auch und dann letzter ergibt sich dann immer

Randomisierte Suche und Escaping

Randomisierte Suche:

⇨ Idee: Grossen N(x) ist aufwendig, um alle Nachbarn zu evaluieren

- Dies kann man vermeiden, wenn man eine Methode zur zufälligen Wahl einer Nachbarlösung verwendet (Stoppen nicht wenn keine Verbesserung mehr sondern nach einer fixen Iterationszahl)

1. Konstruiere eine Startlösung $x \in S$.
2. Für $j = 1, \ldots, \ell$:
   1. Wähle durch eine geeignete Zufallsmethode ein $x' \in N(x)$.
   2. Wenn $f(x') < f(x)$: setze $x \leftarrow x'$.
3. Lösung x ausgeben.

Lokale Suche mit Escaping-Mechanismus:

- Die lokale Suche lässt nur eine Verbesserung zu

⇨ Man möchte vermeiden, dass schlechtes lokales Optimum erreicht wird

⇨ Escaping = Aus lokalem Optimum herauskommen, indem zweitweise Verschlechterung der Lösung x zugelassen wird

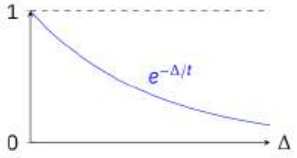
Metropolis Algorithmus (Variante der randomisierten Suche mit Escaping)

1 Konstruiere eine Startlösung $x \in S$ und sei $x^\Delta \leftarrow x$ die bisher beste.
2 Für $j = 1, \ldots, \ell$:
  1 Wähle durch eine geeignete Zufallsmethode ein $x' \in N(x)$.
  2 Wenn $f(x') < f(x)$: setze $x \leftarrow x'$. → Verbesserung
  3 Wenn $f(x') \geq f(x)$ und accept$(f(x') - f(x))$: setze $x \leftarrow x'$. → Keine Verbesserung aber erlaubt
  4 Wenn $f(x') < f(x^\Delta)$: setze $x^\Delta \leftarrow x'$. → Mom. beste
3 Lösung x^Δ ausgeben.

- ⇨ Vermeidet lokale Optima **in der** Nachbarschaft (auch in einer Nachbarschaft sind mehrere lokale Optima möglich => Nachbarschafts Optima = Ein lokales Optima), denn es erlaubt den Wechsel zu schlechteren Lösungen
- ⇨ Damit werden kleinere Nachbarschaften sinnvoll nutzbar
- ⇨ Klassisches Akzeptanzkriterium: Metropolis Kriterium

$$\text{accept}(\Delta) = (e^{-\Delta/t} > r)$$

- mit Normierungsfaktor $t > 0$ und
- der gleichverteilten Zufallszahl $r \in \mathcal{U}[0, 1)$, wobei $0 \leq r < 1$.



- ⇨ Idee: Nah am Optimum = Schritte verkleinern

Simulated Annealing in TSP

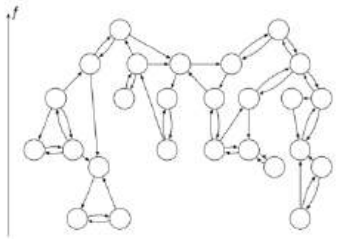
1 Konstruiere eine Startlösung $x \in S$, $x^\Delta \leftarrow x$,
Starttemperatur $t \leftarrow t_0$.
2 Wiederhole solange $t \geq t_{\min}$:
  1 Für $j = 1, \ldots, \ell$:
    1 Wähle durch eine geeignete Methode ein $x' \in N(x)$.
    2 Wenn $f(x') < f(x^\Delta)$: setze $x^\Delta \leftarrow x'$.
    3 Wenn $f(x') < f(x)$: setze $x \leftarrow x'$.
    4 Wenn $f(x') \geq f(x)$ und accept$(f(x') - f(x))$: setze $x \leftarrow x'$.
  2 Setze $t \leftarrow \alpha(t)$.
3 Lösung x^Δ ausgeben.

- ⇨ Unterschied zu Metropolis: Simulated Annealing macht globale Optimierung und Temperatur ist im Metropolis **konstant** (im SA sinkend => Akzeptanz von schlechten Lösung mit abnehmender Wahrscheinlichkeit)
- ⇨ alpha(t) = Abkühlung (geometrische Abkühlung => alpha(t) = a*t)
- ⇨ t_min nahe bei 0 und t0 bspw. 10 Mal so hoch wie höchste potenzielles f(x)-f(x')

Zusammenhängende Nachbarschaft:

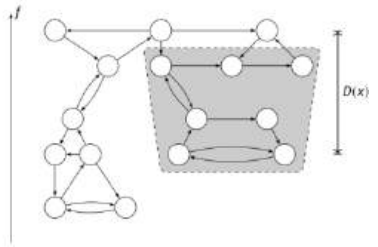
- ⇨ Nachbarschaft = zusammenhängend wenn von jeder Lösung x zu jeder beliebigen anderen Lösung stets ein Weg existiert der von Nachbar zu Nachbar geht
- ⇨ Verschlechterung zugelassen

$$x \in N(x') : x' \in N(x'') : x'' \in \cdots N(\hat{x}).$$



Tiefe eines lokalen Min:

- ⇨ Tiefe D(x) eines lokalen Mins x ist die kleinste Differenz im Funktionswert, die zu überwinden ist, um aus dem lokalen Min herauszukommen:



Theorem von Hajek über Konvergenz:

- ⇨ Gegeben: zusammenhängende Nachbarschaft, Temperatur t monoton gegen 0 konvergiert, Bei einem Weg von x' nach x'' und zurück alle Zwischenlösungen kleiner als ein Wert F sind
- ⇨ Dann gilt: t1…tk = Temperatur, D = maxD(x) = kleinste Tiefe (um aus grössten lokalen Min herauszukommen):

$$\sum_{k=1}^{\infty} e^{-D/t_k} = +\infty.$$

- Nach unendlichen Iterationen wird eine optimale Lösung erreicht.

- ⇨ Bei jedem Schritt genug Möglichkeiten, nicht in einem Tal eingesperrt werden (reversibilität) und Temp langsam genug abkühlt dann Garantie dass irgendwann beste Lösung gefunden wird.
- ⇨ Wenn Summe endlich wäre, dann könnte Verfahren in einem schlechten Min stecken bleiben

Tabu Suche

- ⇨ Escaping von SA kann zyklieren
- ⇨ Zyklus wenn nach einigen Escape Schritte in dasselbe oder ähnliches lokales Optimum zurück geht
- ⇨ Tabu => Verhindert Zyklen durch Tabu-Setzen (Verbieten) von inversen Schritten (Rückweg)
- ⇨ Gedächtnis speichert Informationen über bisherigen Verlauf

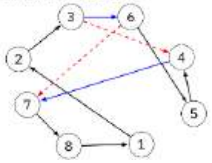
Attributiver Speicher:

- ⇨ Gedächtnis => Speichern aller bisherigen Lösungen speicheraufwändig
- ⇨ Daher nur einzelne Attribute einer Lösung oder Übergangs speichern
- ⇨ Weist eine Nachbarlösung bzw. der Übergang zu ihr Attribut auf aus Tabu-Liste, wird diese Lösung vermieden
- ⇨ Hinweis: Verschiedene Lösungen können dieselben Attribute aufweisen. D.h. nicht bekannte Lösungen können trotzdem vermieden werden.

Beispiel Attribute im TSP:

- tabu-to-drop: Tabu Liste von Kanten (sie dürfen nicht aus einer Lösung gelöscht werden, weil sie für die aktuelle Lösung gerade erst hinzugefügt)
- tabu-to-add: (separate) Tabu Liste von Kante (sie dürfen nicht in eine Lösung hinzugefügt werden, weil sie für aktuelle Lösung gerade erst entfernt wurden)

- Durch 2-OPT bei $i = 3$ und $j = 6$ wurden soeben die Kanten $(3, 4)$ und $(6, 7)$ entfernt. Diese stehen nun auf der *tabu-to-add*-Liste.
- Gleichzeitig wurden die Kanten $(3, 6)$ und $(4, 7)$ hinzugefügt. Diese stehen nun auf der *tabu-to-drop*-Liste.



Algorithmus:

1. Konstruiere eine Startlösung $x \in S$ und sei $x^\Delta \leftarrow x$ die bisher beste.
2. Wiederhole, bis ein Stopp-Kriterium erfüllt ist:
   1. Wähle durch die Best Improvement Methode eine Lösung $x' \in N(x)$, die nicht tabu ist.
   2. Setze $x \leftarrow x'$ und aktualisiere die Tabu-Liste.
   3. Wenn $f(x') < f(x^\Delta)$: setze $x^\Delta \leftarrow x'$.
3. Lösung x^Δ ausgeben.

- Einfaches Stopp-Kriterium: Anzahl Iterationen beschränken.
  - ⇨ Durch «die nicht tabu ist» wird verhindert, den gleichen Zug zu machen wie in einer Iteration vorher.
  - ⇨ Idee: bspw. in 2-OPT mit Escaping wenn die neue Lösung schlechter ist könnte der Algo diese einfach umgekehrt machen und dann, wenn in lokalem Optimum gefangen, einfach wiederholt zwischen ähnlichen Lösungen springen
  - ⇨ 2-OPT hat nur Improving moves die erlaubt werden (sobald keiner mehr dann wird gestoppt), da keine verschlechterung erlaubt keine Rücksprünge => lokales Optimum wird erreicht (oder evtl. sogar global) => Mit Escaping ist jedoch Zyklus möglich

Anspruchskriterien:

- Heben den Tabu-Status gewisser Tabu-Restriktionen auf (vor allem dann notwendig, wenn alle Nachbarn N(x) einer Lösung x tabu sind)

Wichtigste Varianten:

- *aspiration-by-default*: Falls alle Nachbarn tabu sind, wähle Lösung $x' \in N(x)$, die „am wenigsten" tabu ist (oder zufällig ausgewählt wurde).
- *improved-best*: Falls es eine Tabu-Lösung gibt, die besser ist alle bisherigen Lösungen, wähle diese.

Algorithmus mit Anspruchskriterium:

1. Konstruiere eine Startlösung $x \in S$.
2. Wiederhole, bis ein Stopp-Kriterium erfüllt ist:
   1. Wähle durch die Best Improvement Methode ein $x' \in N(x)$,
      - das nicht tabu ist,
      - oder das tabu ist und ein Anspruchskriterium erfüllt.
   2. Setze $x \leftarrow x'$ und aktualisiere die Tabu-Liste.
   3. Wenn $f(x') < f(x^\Delta)$: setze $x^\Delta \leftarrow x'$.
3. Lösung x^Δ ausgeben.

Gedächtnis:

- Kurzzeitgedächtnis: FIFO-Liste (Queue) mit beschränkter Länge (t = Speicherdauer)
- Wenn t überschritten wird, wird ältester Eintrag gelöscht

Welchen Wert für t?

- Guten t-Wert durch Testen herausfinden.
- Abwägen zwischen dem Auftreten von Zyklen und der übermässigen Einschränkung des Lösungsraums
- t kann von der Instanz, von der Nachbarschaftsgrösse und vom Attribut abhängig gemacht werden.

- Erweiterung: Dynamische Speicherdauer: Random (während Suche t zufällig variieren), Reactive (anhand von Beobachtungen t anpassen)
- Mit einem Gedächtnis kann man analysieren, wie häufig ein Attribut vorkommt (wenn xj in guten oder schlechten Lösungen oft einen gewissen Wert hat, scheint dieser attraktiv bzw. unattraktiv zu sein, auch wenn xj viel wechselt, wird der Wert gerne zur Detailverbesserung verwendet (Bin-Packing kleiner Gegenstand zum Auffüllen verwenden)) => Unterstützt die Suche mit einem Erfahrungswert

Unterschied TSP und BBP:

⇨ TSP sind alle Lösungen zulässig, da Bedingung strukturell

⇨ BBP sind nicht alle Lösungen zulässig, es muss immer Kapazität für Gültigkeit geprüft werden

Nutzung des Häufigkeitsgedächtnis:

⇨ xj ist einfach die jte Variable der Lösung (wenn bspw. weiss das der Wert von xj häufig in guten Lösungen vorkommt kann man ihn bspw. fixieren)

- Intensivierung = Bereich des Lösungsraum wird fokussierter betrachtet (xj auf Wert fixieren der in guten Lösungen verwendet wird)
- Diversifikation = Man will gezielt in andere noch wenig untersuchte Bereiche des Lösungsraums gelangen (xj auf Wert fixieren, der noch kaum verwendet wurde => zunächst mit starker Verschlechterung rechnen)

1 Konstruiere eine Startlösung $x \in S$ und sei $x^\triangle \leftarrow x$ die bisher beste.
2 Wiederhole, bis ein Stopp-Kriterium erfüllt ist:
  1 Wähle durch die Best Improvement Methode ein $x' \in N(x)$,
    - das nicht tabu ist,
    - oder das tabu ist und ein Anspruchskriterium erfüllt.
  2 Setze $x \leftarrow x'$ und aktualisiere die Tabu-Liste sowie das Häufigkeitsgedächtnis.
  3 Wenn $f(x') < f(x^\triangle)$: setze $x^\triangle \leftarrow x'$.
  4 Entscheide, ob eine Intensivierungsphase stattfinden soll und setze die Nachbarschaft N entsprechend.
  5 Entscheide, ob eine Diversifikationsphase stattfinden soll und setze die Nachbarschaft N entsprechend.
3 Lösung $x^\triangle$ ausgeben.

Grenzen:

- Wenn Tabu-Suche einen Teilbereich der Tour gut erkundet und optimiert hat, dann enthält die Tabu-Liste genügend Informationen, dass dieser Bereich so beibehalten wird. Jedoch verblast diese «Erinnerung» irgendwann und beginnt dort von Neuem => Man könnte Tabu-Listen in Regionen unterteilen (dann aber schwieriger über Regionen hinweg zu optimieren)

Genetische Algorithmen

Neue Generation entsteht durch wiederholtes Anwenden von (P = Popultation von Lösungen):

1 *Selektion* wählt ein oder zwei Lösungen aus P zur Fortpflanzung.
2 *Kreuzung* kombiniert zwei bestehende Lösungen.
  ▸ reduziert die Diversität der Population.
3 *Mutation* ändert bestehende Lösung.
  ▸ erhöht die Diversität der Population.

⇨ Diversität einer P stellt sicher, dass verschiede Gegenden des Lösungsraums durch Lösungen bzw. Teile von Lösungen abgedeckt sind

Lösung Repräsentationen:

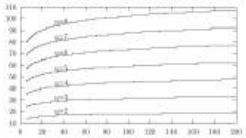
- Binär-String Repräsentation 0, 1, 0, 0, 1, 0, 1, 0,
  z.B. für das Bin-Packing Problem mit genau 2 Behältern: sagt ob ein Gegenstand in Behälter 0 oder 1 ist.
- Integer-String Repräsentation 1, 1, 2, 1, 3, 2,
  z.B. Knotenfärbung: bestimmt die Farbe eines Knotens
- Permutations-Repräsentation 1, 4, 3, 5, 2,
  z.B. Traveling Salesman Problem: gibt die Tour an
  z.B. Knotenfärbung: bestimmt die Knoten-Reihenfolge für die Greedy Heuristik

Init P:

⇨ Typisch: Zufällig initiale Lösung erzeugen (sollte genug Diversität haben, so dass über Teile der Lösungen möglichst der gesamte Suchraum abgedeckt wird)

⇨ Bspw. durch lokale Suche starten (auch wenn bereits konvergiert)

Feste Populationsgrösse setzen, linear bis exponentiell wachsend mit der Problemgrösse.
Theoretischer Wert bei Integer-Strings zur Basis q ($q = 2$ ist binär), dass mit 99.9%-Wahrscheinlichkeit an jeder String-Stelle jede Ziffer mindestens einmal in der Population vorkommt:



Populationsgrösse je String-Länge, siehe Reeves (1993)
String-Länge l

Selektion:

⇨ Wählt eine Lösung aus P

⇨ Für weiterentwickeln von guten Lösungen müssen aus P gute Lösungen gewählt werden (auch schlechte Lösungen interessant, wenn diese Teilaspekte besitzen, die aber unterrepräsentiert sind)

Beispiel Roulette-Rad-Wahl:

⇨ Selektion durch Wahrscheinlichkeit einer Lösung (Lösung => Element in einer P) mittels Maximal

⇨ Minimierung = Kehrwerte

Naives Vorgehen in $O(|P|)$:
1 Gleichverteilte Zufallszahl $r \in [0, 1)$ ziehen.
2 Setze $F \leftarrow 0$.
3 Für jede Lösung $x \in P$:
  1 Setze $F \leftarrow F + f(x)$.
  2 Wenn $r < F / \sum_{y \in P} f(y)$, gib Lösung x aus und stoppe.

⇨ Obiger Algo holt erste (aufsummierte) f(x) Zahl die über r ist

⇨ Besser mittels binärer Suche (Funktionswert in Liste aufkumulieren und dann binäre Suche) => O(log|P|)

⇨ Universelle Wahl:

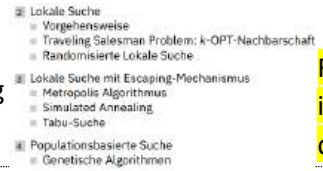
Naives Vorgehen in $O(|P|)$:
1 Gleichverteilte Zufallszahl $r \in [0, 1/a)$ ziehen.
2 Setze $F \leftarrow 0$.
3 Für jede Lösung $x \in P$:
  1 Setze $F \leftarrow F + f(x)$.
  2 Wenn $r < F / \sum_{y \in P} f(y)$, gib Lösung x aus und setze $r \leftarrow r + 1/a$.

Vorteil: deutlich schneller, statistisch genauso gut.

⇨ Mehrere Lösungen gleichzeitig wählen

⇨ Statt mehrmals das Rad zu drehen, drehen wir 1 Mal aber mit mehreren gleichmässig verteilten Messungen

Achtung: Population ist eine Menge von Lösungen (bspw. in TSP eine Menge von Permutationen)
Mutation kann dann jeweils auf einzelne Lösungen angewendet werden (die selektierten Lösungen)
Kreuzung findet oftmals auf allen selektierten Lösungen statt



Für mich als Optimierer bedeutet das No-Free-Lunch-Theorem, dass es keine Garantie gibt, dass meine Optimierung immer gut funktioniert. Ich muss meine Algorithmen stets auf den jeweiligen Problemtypen testen (benchmarken), denn eine Methode, die bei einer Problemklasse gut ist, kann bei einer anderen komplett versagen.

- ⇨ Probleme bei absoluten Zielfunktionswerte: Schnell gehen bis viele ähnlich Lösungen sich durchsetzen, die per Zielfunktion kaum unterscheidbar sind (Vorschnelle Konvergenz)
- ⇨ Verbesserung: Skalierung (auf eine lineare Funktion), Ranking der Lösungen (durch Sortierung => Beste Lösungen zuletzt), Tournament (beste Lösung zufällig aus Teilmenge von Lösungen)

Erzeugung einer neuen Generation:

- ⇨ Selektion wählt aus der aktuellen P Lösungen aus
- ⇨ Über Kreuzung und Mutation wird eine neue P erstellt
- ⇨ Kreuzung findet üblicherweise mit 100% statt (also sicher) und Mutation nur 1% oder 2% (selten => nach Kreuzung)

Genetischer Algorithmus, frei nach Holland (1975)

1. Erzeuge initiale Population P.
2. Wiederhole bis es keine Verbesserung mehr gibt:
   1. Erstelle leere Population $P' \leftarrow \emptyset$
   2. Wiederhole solange $|P'| < |P|$:
      1. Wenn Kreuzung stattfindet: Selektiere Lösungen x, y und führe Kreuzung durch, Resultat in P' einfügen.
      2. Wenn Mutation stattfindet: Selektiere Lösung x und führe Mutation durch, Resultat in P' einfügen.
   3. Setze $P \leftarrow P'$
3. Beste entdeckte Lösung zurückgeben.

Mutation:

- Gegeben: Lösung x durch Selektion

- Bei einer Binär/Integer-Repräsentation:
  m Ziffern werden zufällig ausgewählt (gleichverteilt) und dann
  - Negation der Ziffer (bei Binär-Repräsentation)
  - oder zufällig eine neue Ziffer wählen (bei Integer-Repräsentation).
- Bei einer Permutations-Repräsentation:
  - m-mal zwei Einträge vertauschen (*swap*),
  - m-mal einen Eintrag verschieben (*shift*),
  - Teil-Liste mit m Einträgen selektieren und zufällig permutieren (*scramble sublist*).

⇨ Allenfalls ungültige Lösungen mit bspw. Greedy Verfahren reparieren¨
Beispiel: TSP (P ist eine Permutation von Städten) dann jede Stadt darf genau einmal vorkommen. Mit Mutation können evtl doppelte Städte auftauchen. Dazu kann eine Reparatur gemacht werden.
Bsp. Greedy (immer aktuellen besten Weg gehen => Achtung lokale Suche verändert Lösung während Greedy Lösung konstruiert): Durchlaufe P und ersetze doppelte Elemente mit nicht benutzten Städten

Kreuzung:

- ⇨ Gegeben: Zwei Lösungen x, y (x != y) durch Selektion

Kreuzung bei einer Binär/Integer-Repräsentation der Länge l:

1. Ein Kreuzungs-Punkt k wird aus $\{1, 2, \ldots, l\}$ gezogen.
2. Neue Lösung y' besteht dann aus $y'_i = \begin{cases} x_i, & i \leq k \\ y_i & i > k. \end{cases}$ für $i = 1, \ldots, l$.

Eine weitere Variante ist das Uniform-Crossover:

1. Für Ziffer $\{1, 2, \ldots, l\}$ wird mit 50% der Wert aus Lösung x bzw. aus y verwendet.

- ⇨ Kreuzung bei Permutationen: Problem ist dass Resultat stets eine Permutation ist (neue Lösung)

Mögliche Lösung: Order Crossover (siehe Schöning, 2004):

1. Gegeben Lösungen $x, y \in P$ mit
   $x = (5, 6, 2, 0, 1, 4, 7, 3, 8, 9)$,
   $y = (3, 2, 1, 6, 5, 9, 4, 0, 8, 7)$.
2. Wähle aus x zufällig den Abschnitt $x' = (2, 0, 1, 4, 7, 3)$.
3. Beginne die neue Lösung z mit x'.
4. Dann setze fort in y beim letzten Knoten von x'. Dabei füge nur diejenigen Knoten aus y ein, die in x' noch nicht vorgekommen sind:
   $z = (2, 0, 1, 4, 7, 3, 6, 5, 9, 8)$.

Führen Sie dies für Abschnitt $x' = (5, 6, 2, 0, 1)$ durch.
$z = (5, 6, 2, 0, 1, 9, 4, 8, 7, 3)$

- Weitere mögliche Lösung: Union Crossover (Poon&Carter, 1995):
  1. Gegeben Lösungen $x, y \in P$ mit
     $x = (2, 4, 1, 6, 3, 5)$,
     $y = (5, 1, 3, 2, 6, 4)$.
  2. Wähle aus x zufällig den Abschnitt $x' = (4, 1, 6)$.
  3. Setze y' auf die übrigen Elemente $y \setminus x'$, also $y' = (5, 3, 2)$.
  4. Erzeuge leere, neue Lösung $z = ()$
  5. Wiederhole solange $x' \neq ()$ und $y' \neq ()$.
     1. Wähle zufällig x' oder y'.
     2. Entferne daraus den ersten Eintrag und hänge ihn an z an.
  6. Hänge x' und y' an z an und gib z zurück.

Führen Sie die Schleife aus und notieren Sie das Ergebnis.
Alg. mit x' start
$z = (4, 5, 1, 3, 6, 2)$

No Free Lunch (NFL) Theorem

- Welche Metaheuristik ist insgesamt die beste?
- Gemäss NFL sind alle metaheuristiken ungefähr gleich gut (sofern Lösungen nicht doppelt besucht) => Wenn Heuristik in Problem gut abschneidet dann schneidet sie bei anderen Problemen schlecht ab (hebt sich auf)
- Per Zufall konstruierte Lösungen sind also im Durchschnitt genau so gut wie Verbesserungsheuristiken
- Es gibt kein Allheilmittel in der Optimierung. Jeder Algorithmus ist nur so gut wie seine Passung zum Problem.

⇨ In der lokalen Suche ist die Nachbarschaft einer Lösung die Menge aller Lösungen, die man durch kleine, definierte Änderungen an der aktuellen Lösung direkt erreichen kann.

B&B ist eine Suche nach optimaler Lösung (ein Lösungsraum muss daher bereits existieren).
DP erstellt schrittweise (konstruktiv) eine optimale Lösung mittels Memoisierung, die genutzt werden kann, wenn Teilprobleme / Strukturen überlappend sind (eine optimale Lösung eines Problems enthält optimale Lösungen seiner Teilprobleme => Dann funktioniert DP) => Optimale Teilproblem notwendig für Korrektheit, Überlappung notwendig für Effizienz

Exakte Algorithmen (Beispiele mit TSP, BPP)

Einleitung

Idee: Suche zur exakten Lösung eines Problems

- ⇨ Daraus entsteht das Problem von exponentiell grossen Suchbäumen
- ⇨ Ein Problem, für das es einen Suchbaum gibt kann auch verschiedene Arten gelöst werden:

- Vollständige Enumeration
  - Sämtlichen Lösungen werden betrachtet.
  - Durchsuchen z.B. eines kompletten Suchbaums per Tiefensuche.
- Dynamische Programmierung
  - Speichert Lösung von Teilproblemen.
  - Vermeidet die erneute Lösung sich wiederholender Teilprobleme.
- Branch & Bound Suche
  - Gegeben eine Teillösung (die durch *Branching* erzeugt wurde), schätzt man durch *Bounding* ab, wie gut der Rest des Problems im besten Fall gelöst werden kann.
  - Ist der Best-Case für die Teillösung garantiert schlechter als die aktuell beste Lösung für das Gesamtproblem, dann kann diese Teillösung nicht zu einem Optimum führen und wird verworfen.
- Manchmal ist eine Kombination der Verfahren möglich.

Dynamische Programmierung für TSP

- ⇨ Grundprinzip ist das Teilen des Problems in kleinere Teilprobleme und durch Memoisierung Teillösungen merken

- Setze $C(\{\}) \leftarrow 0, L(\{\}) \leftarrow 0$.
- Für $i = 1, \ldots, n$:
  1. Für jede Knotenmenge $S \subseteq V$ aus $|S| = i$ Knoten:
     1. Setze $j \leftarrow \arg\min_{j \in S} C(S \setminus \{j\}), 0) + c(L(S \setminus \{j\}), j) + c(j, 0)$
     2. Setze $L(S) \leftarrow j$
     3. Setze $C(S) \leftarrow C(S \setminus \{j\}) - c(L(S \setminus \{j\}), 0) + c(L(S \setminus \{j\}), j) + c(j, 0)$
- $C(V)$ ist nun der optimale Zielfunktionswert.

Beispiel:

Sei folgende Kostenmatrix gegeben:

	0	1	2	3
0	∞	10	15	20
1	10	∞	35	25
2	15	35	∞	30
3	20	25	30	∞

(∞ = Kein Weg zu sich selbst (nicht relevant TSP))

- ⇨ Ziel: Min. Rundreise, die in Stadt 0 startet, alle anderen Städte genau 1 Mal besucht und zu Stadt 0 zurückkehrt.
- ⇨ c(S,j): Min kosten, um von Stadt 0 über alle Städte in S zu reisen und in Stadt j zu enden

1. Setze C({}) und L({}) = 0
2. i = 1, |S| = 1:

S	j	c(S,j)
{1}	1	10
{2}	2	15
{3}	3	20

3. i = 2, |S| = 2:

S	j	c(S,j)	Wert
{1,2}	1	c({2},2)+d(2,1) (15+50)	50
	2	c({1},1)+d(1,2)	45
{1,3}	2	c({3},3)+d(3,1)	45
	3	c({1},1)+d(1,3)	35
{2,3}	2	c({3},3)+d(3,2)	50
	3	c({2},2)+d(2,3)	45

- ⇨ c(S, j) = min über k ∈ S \ {j} von [c(S \ {j}, k) + d(k, j)
- ⇨ …
- ⇨ Es geht darum das TSP enumerativ zu lösen, jedoch Teilkalkulationen nicht mehr doppelt zu machen, sondern aus Cache zu holen (DP)

Branch & Bound Suche

- ⇨ Zerlegung eines Problems in Teilprobleme
- ⇨ Die möglichen Lösungen S ' eines Teilproblems Π ' bilden eine Teilmenge S ' ⊂ S aller zulässigen Lösungen
- ⇨ Ob sich in S ' eine optimale Lösung verbirgt ist unklar

Branching

branches(x') ermittelt für eine Teillösung x' bzw. Teilproblem Π' eine Menge von Teillösungen $\{x'_i\}$ bzw. von Teilproblemen $\{\Pi'_i\}$.
Die zugehörigen Mengen möglicher Lösungen $\{S'_i\}$ sind paarweise disjunkt und umfassen alle möglichen Lösungen S'' von Teilproblem Π'.

- ⇨ Es ergibt sich ein Suchbaum mit Traversierung via bspw. Tiefensuche, Breitensuche (hat grossen Speicherbedarf => Kein vergessen wie bei Tiefensuche, muss alle Knoten auf einer Ebene speichern => Wächst exponentiell) oder Bestensuche (erfordert eine approximative Bewertungsfunktion)
- ⇨ Nach einem Branching ist die Frage ob die Lösung eines Teilproblems zu einem globalen Optimum kommt => Falls beweisbar, wird Problem nicht wieder betrachtet (Beweis = durch Bounding)

Grundlegende B&B Idee:

Während des Verfahrens wird jeder Branch untersucht und berechnet eine untere Schranke für diesen Teilraum
Wenn die untere Schranke grösser ist als bisher bekannte obere Schranke, kann man diesen Branch verwerfen => weil keine bessere Lösung in diesem Branch existieren kann (Beweis) => Obere Schranke wird immer aktualisiert, sobald eine bessere zulässige Lösung gefunden wurde

Bounding

- Funktion *bound*(*x*) ermittelt den besten Zielfunktionswert, den man durch Komplettieren der zu Π' gehörenden Teillösung *x* noch erreichen kann.
- Ist die Bound schlechter als der bisher beste bekannte Zielfunktionswert, dann kann durch Komplettierung von *x* keine global optimale Lösung erreicht werden.

⇨ Worst Case = Laufzeit entspricht Suchbaumgrösse (d.h. bounding schlägt immer fehl und Teilraum muss jeweils weiter betrachtet werden)

⇨ = Schranken

Konkreter Ablauf:

Branch&Bound Tiefensuche

1. Setze $f^\wedge$ auf einen ungültigen Wert. Optional: durch Heuristik eine bisher beste Lösung $x^\wedge$ bestimmen und $f^\wedge \leftarrow f(x^\wedge)$ setzen.
2. Sei $x \leftarrow \emptyset$ eine leere Teillösung.
3. Rufe *b&b-tiefensuche*(*x*) auf.
4. Optimale Lösung $x^* \leftarrow x^\wedge$ ausgeben.

function *b&b-tiefensuche*((Teil-)Lösung *x*) >= Blatt

- Falls *x* eine vollständige Lösung ist:
  - Falls $f(x)$ besser ist als $f^\wedge$, dann $x^\wedge \leftarrow x$ und $f^\wedge \leftarrow f(x^\wedge)$ setzen.
- Sonst: Für jede (Teil-)Lösung $x' \in branches(x)$:
  - Falls *bound*(x') besser ist als $f^\wedge$:
    - Rufe *b&b-tiefensuche*(x') auf.

⇨ bound(x) gibt true zurück wenn besser als beste Lösung (d. h. es gibt eine Chance, dass die aktuelle Teillösung besser ist, als die momentane optimale Lösung)

⇨ f delta = obere Schranke (Zielfunktionswert)

Beispiel am Bin Packing Problem (Ablauf):

Bin Packing Problem

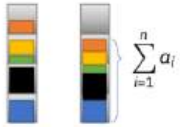
Gegeben: *n* Gegenstände mit der Grösse $a_1, \ldots, a_n \in (0, 1]$.
Gesucht: eine Zuordnung in Behälter (Bins) $B_1, B_2, \ldots, B_m$ sodass $\sum_{a_i \in B_k} a_i \leq 1$ für jeden Behälter $k = 1, \ldots, m$ gilt, sodass die Anzahl genutzter Behälter *m* minimiert wird.

function *bpp-b&b-dfs*((Teil-)Lösung *x* mit Gegenstand *i*)

- Falls $i > n$: Aktuelle Lösung ausgeben.
- Falls $i \leq n$: Für Behälter $k = 1, \ldots, n$:
  - Falls Gegenstand *i* in Behälter *k* passt:
    - Gegenstand *i* in Behälter *k* packen.
    - Falls *bound*(aktuelle Lösung) $< f^\wedge$: Rufe *bpp-b&b-dfs*($i + 1$) auf.
    - Gegenstand *i* aus Behälter *k* entfernen.

⇨ Es braucht maximal n Behälter (daher Loop von Anzahl Behälter in dfs)

⇨ Eine Teillösung x = [i, m, (c(1),…,c(m)] wird repräsentiert durch: Nächster betrachtender Gegenstand i, die Anzahl genutzter Behälter m, die Restkapazitäten der Behälter c(1),…,c(m)



- $\lceil \sum_{i=1}^{n} a_i \rceil$ ist eine untere Schranke für die minimale Anzahl Behälter.
- Für eine (Teil-)Lösung $x = [i, m, (c(1), \ldots, c(m))]$ mit $i = 1, \ldots, n+1$ kann man diese untere Schranke verbessern (also erhöhen) zu

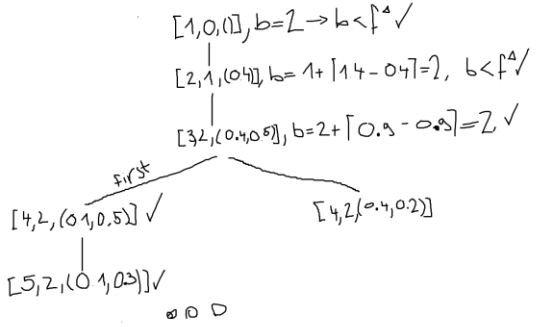
$$bound(x) = m + \max\left\{0, \left\lceil \underbrace{\left(\sum_{i'=i}^{n} a_{i'}\right)}_{\text{Summe Restgrössen}} - \underbrace{\left(\sum_{k=1}^{m} c(k)\right)}_{\text{Summe Restkapazitäten}} \right\rceil\right\}$$

⇨ Wenn Restgrössen < Restkapazität, bedeutet dies, dass es noch möglich ist, diese alle in m Behältern zu platzieren, daher ist bound(x) dann nur m + 0, da die Teillösung in m Behältern sein kann. Wenn > dann bedeutet dass, dass der Platz in den m Behältern sowieso nicht mehr ausreicht und es braucht mindestens soviel Behälter mehr, wie die die volle aufgefüllten Kapazität der m Behälter + der restlichen Restgrösse (aufgerundet da Anzahl Behälter eine natürliche Zahl).

Konkretes Beispiel mit [1, 0, ()]:

i	1	2	3	4	5	6
a_i	0.6	0.5	0.3	0.2	0.2	0.2
$\sum_{i' \geq i} a_{i'}$	2.0	1.4	0.9	0.6	0.4	0.2

Wenden Sie die Branch&Bound Tiefensuche an.
- Setzen Sie $f^\wedge \leftarrow \infty$ und rufen Sie *bpp-b&b-dfs*([1, 0, ()]) auf.
- Notieren Sie in jedem Schritt die (Teil-)Lösung *x* und *bound*(*x*).



⇨ **Achtung:** Es muss bei jedem Knoten durch alle möglichen Behälter durch geloopt werden (auch leere bis n-Gegenstände, da max). Man kann jeweils auch sagen, dass keine Behälter dazwischen leer sein dürfen => So spart man unnötige Berechnungen.

⇨ Sobald Blattknoten erreicht wird f_delta überschrieben.

⇨ Immer zuerst in die Tiefe gehen (quasi so schnell wie möglich eine Lösung bekommen)

⇨ Das loopen durch alle Behälter bei jedem Knoten ist notwendig um alle möglichen Lösungen anzuschauen.

⇨ Sobald bound(x) >= f_delta dann schneide ab (brich Teillösung ab)

Beispiel am Traveling Salesmen Problem Ablauf:

Branching:

Branching durch Kantenwahl

Für jede wählbare (Teilzyklen-vermeidende) Kante (i, j) ausgehend vom aktuellen Knoten i:

- Entferne Zeile i,
- Entferne Spalte j,
- Entferne Rückwärtskante (j, i) durch Setzen auf „–".

Beispiel:

	1	2	3	4	5
1	–	5	13	28	17
2	5	–	9	4	14
3	13	9	–	6	7
4	28	4	6	–	11
5	17	14	7	11	–

⟹ Wahl (1,2)

(1,2)	1	3	4	5
2	–	9	4	14
3	13	–	6	7
4	28	6	–	11
5	17	7	11	–

Bounding:

Bounding durch Summe der Zeilenminima

bound(M): Minimum der Kantengewichte in jeder Zeile berechnen und aufsummieren, plus die schon bestehende Tour.

- Eine optimale Rundreise durch die Matrix enthält mindestens eine Kante aus jeder Zeile.
- Damit ist eine optimale Rundreise mindestens so lange.
- Somit ist *bound*(M) eine untere Schranke für die Länge jeder Tour, die noch aus der vorliegenden Matrix entstehen kann.

Beispiel: $M =$

	1	2	3	4	5
1	–	5	13	28	17
2	5	–	9	4	14
3	13	9	–	6	7
4	28	4	6	–	11
5	17	14	7	11	–

, *bound*$(M) = 26$.

⇨ Zeilen repräsentieren Städte und Startpunkte und Spalten Destinationen

⇨ Die untere Schranke muss noch keine optimale Rundreise sein (das ist ja das Ziel, was man durch den Algo erreichen will)

	1	2	3	4	5
1	–	5	13	28	17
2	5	–	9	4	14
3	13	9	–	6	7
4	28	4	6	–	11
5	17	14	7	11	–

Die Tour ist leer und beginnt in Knoten 1.
Ausgehend von Knoten 1 gibt es ein Branching zu den Kanten (1, 2), (1, 3), (1, 4), (1, 5).

(1,2)	1	3	4	5
2	–	9	4	14
3	13	–	6	7
4	28	6	–	11
5	17	7	11	–

bound = (4 + 6 + 6 + 7) + 5 = 28
► Wird als nächstes untersucht.

(1,3)	1	2	4	5
2	5	–	4	14
3	–	9	6	7
4	28	4	–	11
5	17	14	11	–

bound = (4 + 6 + 4 + 11) + 13 = 38

(1,4)	1	2	3	5
2	5	–	9	14
3	13	9	–	7
4	–	4	6	11
5	17	14	7	–

bound = (5 + 7 + 4 + 7) + 28 = 51

(1,5)	1	2	3	4
2	5	–	9	4
3	13	9	–	6
4	28	4	6	–
5	–	14	7	11

bound = (4 + 6 + 4 + 7) + 17 = 38

(1,2)	1	3	4	5
2	–	9	4	14
3	13	–	6	7
4	28	6	–	11
5	17	7	11	–

Die bestehende Tour hat die Länge 5 und endet in Knoten 2.
Ausgehend von Knoten 2 gibt es ein Branching zu den Kanten (2, 3), (2, 4), (2, 5).

(2,3)	1	4	5
3	13	6	7
4	28	–	11
5	17	11	–

bound = (4 + 11 + 11) + 9 + 5 = 42

(2,4)	1	3	5
3	13	–	7
4	28	6	11
5	17	7	–

bound = (7 + 6 + 7) + 4 + 5 = 29

► Wird als nächstes untersucht, Grund: heuristische Wahl der Distanzmatrix mit kleinster Bound.

(2,5)	1	3	4
3	13	–	6
4	28	6	–
5	17	7	11

bound = (6 + 6 + 7) + 14 + 5 = 38

Polyeder: Nicht möglich (sonst wären auch genau 2 Lösungen in LP möglich) Konvexität muss gewährleistet bleiben d.h. wenn A und C zulässig dann alles dazwischen in einer geraden Linie auch zulässig (V mit Ungleichungen geht nicht)

Es gilt: Jeder Halbraum ist Konvex, Der Schnitt von konvexen Mengen ist wieder Konvex

Eine Menge M ist konvex, wenn für jede zwei Punkte x,y∈M auch alle Punkte auf der Strecke zwischen x und y in M liegen

Lineare Programme (LP)

Einleitung

- ⇨ Heuristiken & Metaheuristiken = **wie** finde ich die Lösung (suche nach guten zulässigen Lösungen und werden verwendet falls keine exakte Methode existiert bzw. nicht effizient), keine Optimalitätsgarantie
- ⇨ Mathematische Modellierung mittels Integer Linear Programming (ILP) = **was** muss eine Lösung erfüllen (suche nach optimalen Lösung)

Allgemeines Optimierungsproblem

$$\Pi : \min\{f(\boldsymbol{x}) \,|\, \boldsymbol{x} \in S\}, \quad \text{wobei } S \subseteq \mathbb{R}^n$$

- Entscheidungsvariablen: $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)^\top \in \mathbb{R}^n$
- zulässige Menge: $S \subseteq \mathbb{R}^n$
- Zielfunktion: $f : S \to \mathbb{R}$
- **zulässige Lösung:** $\boldsymbol{x} \in S$
- **optimale Lösung:**

$$\boldsymbol{x}^* \in S \text{ so dass } f(\boldsymbol{x}^*) \leq f(\boldsymbol{x}) \text{ für alle } \boldsymbol{x} \in S$$

- **Optimum, optimaler Zielfunktionswert:** $f(\boldsymbol{x}^*)$

- ⇨ Entscheidungsvariablen: Dinge die man beeinflussen kann (Menge von Material A, Wieviel Arbeitszeit etc.)
- ⇨ Zulässige Menge S: Möglichkeiten die erlaubt sind für Entscheidungsvariablen (bspw. nicht negativ viel Material etc.)
- ⇨ Zielfunktion f(x): Das Ziel das wie optimieren möchten (bspw. Gesamtkosten minimieren)
- ⇨ Zulässige Lösung: Jede Entscheidung x, die alle Bedingungen erfüllt (also in der zulässigen Menge liegt) ist eine zulässige nicht unbedingt optimale Lösung
- ⇨ Optimal Lösung x*: Beste zulässige Entscheidung (die, bei der wir das Ziel (bspw. Minimierung der Zielfunktion) am besten erreichen)

Transport Problem Beispiel

Firma XY produziert in Basel und Zürich ein bestimmtes Produkt. An einem Tag werden 14 Tonnen in Basel und 16 Tonnen in Zürich produziert. Die Firma hat drei Abnehmer, die sich in Luzern, Chur und Winterthur befinden. Der tägliche Bedarf der Abnehmer ist 9 Tonnen für Luzern, 10 Tonnen für Chur und 11 Tonnen für Winterthur.

Firma XY möchte ihre Transportkosten minimieren, deshalb hat sie die Transportpreise pro Tonne auf allen Verbindungen erhoben. Die Ergebnisse sind in folgender Tabelle zusammen getragen:

	Luzern	Chur	Winterthur
Basel	11	4	5
Zürich	5	4	7

Wie soll die Firma XY ihre täglichen Transporte planen, damit die täglichen Transportkosten so klein wie möglich sind?

Parameter:
Matrix als Variablen
Variablen:
xb_l, xb_c, xb_w, xz_l, xz_x, xz_w (Wie viel Produkte für welche Abnehmer)
Zielfunktion:
11 * xb_l + 4 * xb_c + 5 * xb_w + 5 * xz_l + 4 * xz_x + 7 * xz_w + -> min
Nebenbedingung:
xb_l + xb_c + xb_w <= 14
xz_l + xz_x + xz_w <= 16
xb_l + xz_l = 9
xb_c + xz_c = 10
xb_w + xz_w = 11, x… >= 0

Produktionsproblem Beispiel

Beispiel: (Guenin et al. S. 2 – 4)[1]
Firma WaterTech produziert 4 Produkte. Dabei werden 2 Maschinen und 2 Arten Arbeit (trainiert und untrainiert) benötigt. In der folgenden Tabelle findet man Angaben zur benötigten Maschinenzeit und Arbeitszeit, die nötig ist, um 1 Stück jedes Produkts zu erzeugen. Ausserdem sind dort die Verkaufspreise der Produkte angegeben.

Produkt	Maschine 1	Maschine 2	trainierte Arbeit	untrainierte Arbeit	Preis
1	11	4	8	7	300
2	7	6	5	8	260
3	6	5	5	7	220
4	5	4	6	4	180

Die Maschine 1 kann 700 Stunden im Monat und Maschine 2 kann 500 Stunden im Monat betrieben werden. Ausserdem kann WaterTech höchstens 600 Stunden trainierter Arbeit zu $8 pro Stunde und höchstens 650 Stunden untrainierter Arbeit zu $6 pro Stunden einkaufen.

Wie viel Stück von den 4 Produkten soll pro Monat produziert werden, damit der Gewinn maximal wird?

Parameter:
p1, p2, p3, p4 (pi = Preis von Produkt i)
… (Matrix)
Variablen:
x1, x2, x3, x4 (xi = Menge von Produkt i pro Monat)
y1, y2
Zielfunktion:
p1 * x1 + p2 * x2 + p3 * x3 + p4 * x4 – 8 * y1 – 6 * y2 -> max
Nebenbedingungen:
11 * x1 + 7 * x2 + 6 * x3 + 5 * x4 <= 700
4 * x1 + 6 * x2 + 5 * x3 + 4 * x4 <= 500
y1 <= 600, y2 <= 650
y1 = 8 * x1 + 5 * x2 + 5 * x3 + 6 * x4
y2 = 7 * x1 + 8 * x2 + 7 * x3 + 4 * x4, x.., y.. >=0, x... in Z

Begriffe

Definition: Lineare Funktion

Eine **Funktion** $f : \mathbb{R}^n \to \mathbb{R}$ heisst **linear**, falls $f(\boldsymbol{x}) = \boldsymbol{a}^\top \boldsymbol{x}$, wobei $\boldsymbol{a} \in \mathbb{R}^n$.

Definition: Lineare Nebenbedingung

Eine **Nebenbedingung** heisst **linear**, falls sie eine der folgenden Formen hat

- $f(\boldsymbol{x}) \leq \beta$
- $f(\boldsymbol{x}) \geq \beta$
- $f(\boldsymbol{x}) = \beta$

wobei f eine lineare Funktion ist und $\beta \in \mathbb{R}$.

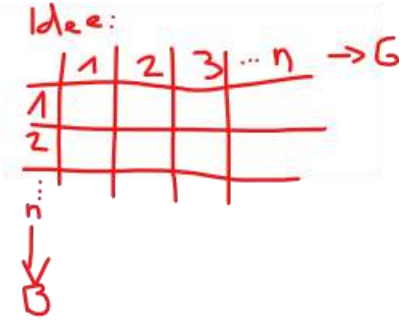
Definition: Lineares Programm (LP)

Das Problem der Minimierung/Maximierung einer linearen Funktion, so dass eine endliche Anzahl von linearen Nebenbedingungen erfüllt ist, heisst **lineares Programm**.

Bin Packing Problem Beispiel

Seien n Gegenstände gegeben, deren Grösse jeweils maximal 1 ist. Wir möchten diese Gegenstände in mehrere Behälter der Grösse 1 packen. Wie viele Behälter brauchen wir dafür mindestens?

Im ersten Teil der Vorlesung haben Sie die First Fit Heuristik kennen gelernt, die eine zulässige Lösung des Problems liefert. Nun wollen wir dieses Problem mathematisch modellieren.



⇨ Matrix mit 0, 1 als Komponenten

Parameter:

ai: Grössen der Gegenstände

Variablen:

xij für i = 0..n und j=0..n (max. Anzahl Bins = Anzahl Gegenstände) und xij in {0,1} => sagt ob Gegenstand i in Box j

yj => sagt ob Box j benutzt wird

Zielfunktion:

Sum(j=1 bis n) yj -> min

Nebenbedingungen:

- Jeder Gegenstand genau in einem Bin
  sum(j=0 bis n) xij = 1, für i = 0..n
- Jeder Bin Kapazität = 1
  sum(i=1 bis n) ai*xij <= yj

Zuordnungsproblem Beispiel

Beispiel: (Guenin et al: S. 15 – 17)[1]
Firma WaterTech hat 4 Jobs zu vergeben. Diese sollen von 4 Mitarbeitern ausgeführt werden. Aus Erfahrung weiss die Firma, wie lange ein bestimmter Mitarbeiter für einen bestimmten Job braucht. Diese Informationen sind in folgender Tabelle aufgeführt:

Job / MA	1	2	3	4
1	3	5	1	7
2	8	2	2	4
3	2	1	6	8
4	8	3	3	2

Zusätzlich soll gelten:

- jedem Mitarbeiter soll genau ein Job zugewiesen werden
- jeder Job soll genau einem Mitarbeiter zugewiesen werden

Wie sollen die Jobs den einzelnen MA zugeteilt werden, damit die Gesamtzeit, die für die Ausführung der Jobs notwendig ist, minimal wird?

Variablen:

I = 0..4 => Menge von Mitarbeitern

J = 0..4 => Menge von Jobs

xij = {0,1} => Mitarbeiter i macht Job j?

Zielfunktion:

aij * xij (für alle i und j) -> min

Nebenbedingungen:

sum(i = 0 bis n) xij = 1 (Jeder Mitarbeiter in genau einem Job)

sum(j = 0 bis n) xij = 1 (Jeder Job genau einen Mitarbeiter)

xij = {0,1}

LP-Formen

Standardform

$$\max\{\boldsymbol{c}^\top \boldsymbol{x} \mid \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b},\ \boldsymbol{x} \geq \boldsymbol{0}\}$$

Kanonische Maximierungsform

$$\max\{\boldsymbol{c}^\top \boldsymbol{x} \mid \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b},\ \boldsymbol{x} \geq \boldsymbol{0}\}$$

Kanonische Minimierungsform

$$\min\{\boldsymbol{c}^\top \boldsymbol{x} \mid \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b},\ \boldsymbol{x} \geq \boldsymbol{0}\}$$

⇨ Notwendig für Solver (manche Solver verwenden eine gewisse standardisierte Form)

- Ungleichung $\leq$ zu Ungleichung $\geq$:
  $\boldsymbol{a}^\top \boldsymbol{x} \leq b \quad \leftrightarrow \quad -\boldsymbol{a}^\top \boldsymbol{x} \geq -b$
- Gleichung zu Ungleichungen:
  $\boldsymbol{a}^\top \boldsymbol{x} = b \quad \rightarrow \quad \boldsymbol{a}^\top \boldsymbol{x} \leq b,\ \boldsymbol{a}^\top \boldsymbol{x} \geq b$
- Ungleichungen zu Gleichungen:
  $\boldsymbol{a}^\top \boldsymbol{x} \leq b \quad \rightarrow \quad \boldsymbol{a}^\top \boldsymbol{x} + s = b,\ s \geq 0$
  $\boldsymbol{a}^\top \boldsymbol{x} \geq b \quad \rightarrow \quad \boldsymbol{a}^\top \boldsymbol{x} - s = b,\ s \geq 0$
- Nichpositive Variable zur nichtnegativen Variable:
  $x_j \leq 0 \quad \rightarrow \quad \overline{x_j} := -x_j,\ \overline{x_j} \geq 0$
- Freie Variable zu nichtnegativen Variablen:
  x_j frei $\quad \rightarrow \quad x_j = x_j^+ - x_j^-,\ x_j^+, x_j^- \geq 0$

Betrachten wir folgendes lineares Programm:

$$\begin{aligned} x_1 + x_2 &\to \max \\ -x_1 + 2x_2 &\leq 8 \\ 2x_1 + x_2 &\leq 14 \\ 2x_1 - x_2 &\leq 10 \\ x_1, x_2 &\geq 0 \end{aligned}$$

- In welcher Form liegt das LP vor? Wie lautet die entsprechende Matrix A und die Vektoren b, c?
- Überführen Sie dieses LP in die Standardform. Wie ändern sich A, b und c?
- Schreiben Sie das LP in der Form $\max\{c^\top x \mid Ax \leq b\}$. Wie müssen A, b und c definiert werden?

ok. max. => $A = \begin{pmatrix} -1 & 2 \\ 2 & 1 \\ 2 & -1 \end{pmatrix}$, $b = \begin{pmatrix} 8 \\ 14 \\ 10 \end{pmatrix}$, $c = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$

Standardform:

-x1 + 2x2 + s1 = 8

2x1 + x2 + s2 = 14

2x1 - x2 + s3 = 10

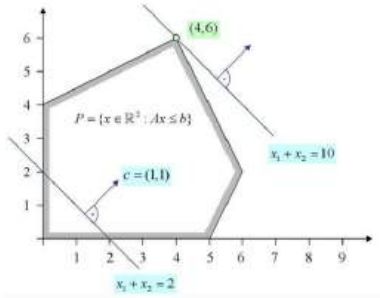
x1, x2, s1, s2, s3 >= 0

A => Einfach noch Einheitsmatrix mit n=3 anhängen

c=(1,1,0,0,0) (=> Zielfunktion kommt s nicht vor und nur einmal x1 und x2)

Graphische Lösung eines LPs

$$\begin{aligned} x_1 + x_2 &\to \max \\ -x_1 + 2x_2 &\leq 8 \\ 2x_1 + x_2 &\leq 14 \\ 2x_1 - x_2 &\leq 10 \\ x_1, x_2 &\geq 0 \end{aligned}$$



Definition

Ein **Polyeder** $P \subseteq \mathbb{R}^n$ ist ein Durchschnitt von endlich vielen Halbräumen, d.h. $P = \{x \in \mathbb{R}^n \mid Ax \leq b\}$.

- ⇨ Polyeder Ränder entsprechen den Nebenbedingungsungleichungen und die Niveaukurven (in Richtung von c) der Zielfunktion mit entsprechendem «y»-Wert
- ⇨ n = Anzahl Entscheidungsvariablen (ohne Schlupfvariablen s, welche künstlich eingefügt wurden um in eine Standardform umzuwandeln)
- ⇨ Polyeder ist der Schnitt von endlich vielen Halbräumen (Menge aller Punkte die eine lineare Ungleichung erfüllen)

Lösungsmengen:

- LP kann, muss aber nicht eine optimale Lösung besitzen. Es gibt drei Fälle:

Definition

Ein LP heisst **zulässig,** falls $P \neq \emptyset$, d.h. es gibt eine zulässige Lösung).
Ein LP heisst **unzulässig,** falls $P = \emptyset$, d.h. es gibt keine zulässige Lösung.
Ein LP heisst **unbeschränkt,** falls es für jede Zahl $\alpha \in \mathbb{R}$ eine zulässige Lösung $x^\alpha \in P$ gibt mit $c^\top x^\alpha \geq \alpha$ (im Fall einer Maximierung).

- ⇨ Genau eine optimale Lösung existiert wenn die Niveaulinie in Richtung c genau einen Eckpunkt erreicht
- ⇨ Unendlich viele wenn die Zielfunktionskurve parallel ist zu einer Kante (Beispiel: max{x1+x2 |x1 + x2 <= 5, x1,x2 >= 0} die ganze Strecke auf Rand x1+x2=5 ist optimal)
- ⇨ Keine Lösung, wenn Polyeder leer (unzulässig) => Kein Schnittpunkt der Halbräume oder Polyeder offen in Richtung von c (Zielfunktion kann immer weiter steigen = keinen max Wert)

Lösbarkeit von LPs:

Satz

Jedes LP erfüllt genau eine der folgenden Möglichkeiten:

- Das LP ist unzulässig
- Das LP ist unbeschränkt
- Das LP besitzt genau eine optimale Lösung
- Das LP besitzt unendlich viele optimale Lösungen

Satz

Es sei ein LP mit der zulässigen Menge P gegeben. Zusätzlich gelte Folgendes:

(i) Das Polyeder P besitzt mindestens eine Ecke
(ii) Das LP besitzt mindestens eine optimale Lösung

Dann ist mindestens eine Ecke des Polyeders P optimal.

Lösungsmethoden für LPs

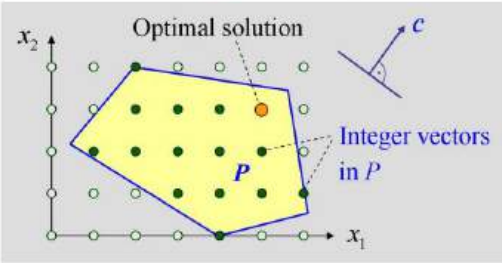
- Simplex-Algorithmus (Dantzig, 1947)
  - Die Methode "springt" von einer Ecke des Polyeders zur benachbarten Ecke, solange sich der Zielfunktionswert nicht verschlechtert, bis sie die optimale Ecke gefunden wird.
- Ellipsoidmethode (Khachiyan, 1979, polynomiell!)
  - keine praktische Bedeutung
- Innere-Punkt-Verfahren (Dikin 1967, Karmarkar 1984)
  - Der Algorithmus bewegt sich im Innern des Polyeders auf dem sog. zentralen Pfad gegen den Rand des Polyeders, wo er deine optimale Lösung findet.

⇨ Simplex ist eine Suche im Lösungsraum entlang der Ecken

Ganzzahlige Lineare Programme (ILP)

Definition

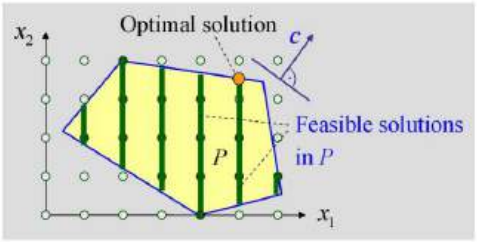
$\max\{c^\top x \mid x \in P \cap \mathbb{Z}^n\}$ mit $P = \{x \in \mathbb{R}^n \mid Ax \leq b\}$



- ⇨ Polyeder ist immer noch gegeben im R-Raum (somit auch Nebenbedingungs-ungleichungen)
- ⇨ Sind NP-vollständig (exponentiell) (LPs sind polynomiell lösbar)

Varianten von Ganzzahligem Programm

- **Binäres** lineares Programm (Binary Linear Program)
  $\max\{c^\top x \mid x \in P \cap \{0,1\}^n\}$ mit $P = \{x \in \mathbb{R}^n \mid Ax \leq b\}$
- **Gemischt-ganzzahliges** lineares Programm (Mixed Integer Linear Program)
  mindestens eine Variable darf nur ganzzahlige Werte annehmen



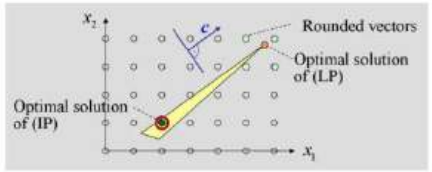
Lösen von ILPs

Naive Variante: (funktioniert nicht!)

- Löse das entsprechende LP (ignoriere Ganzzahligkeit)
- Falls Lösung gebrochene Komponenten enthält, runde auf / ab

⇨ Auf- / Abrunden funktioniert nicht, da dies evtl. keine zulässige Lösung liefert (Nebenbedingung zerstören)

⇨ Optimale Lösung des LPs (und dessen Zielfunktionswert) können beliebig weit von optimalen Lösung des ILPs (und dessen Zielfunktionswert) entfernt sein:

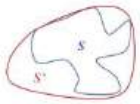


⇨ In der gelben Fläche (Polyeder) ist nur ein diskreter Punkt (von IP) vorhanden, welcher daher optimal ist. Die gerundete LP Version auf einen diskreten Punkt bringt uns keine zulässige Lösung.

⇨ Runden bei binären Problemen kann alle 0-1-Kombis produzieren (Info bzgl. optimalen Lösung vom LP geht verloren)

Besser durch Relaxierung von Problemen:

- Vergrösserung des Lösungsraums (Lösen ohne Ganzzahligkeit)



Proposition
Sei $S \subseteq S'$. Dann gilt:
$\max\{f(x) \mid x \in S\} \leq \max\{f(x) \mid x \in S'\}$

Definition: Relaxierung eines Optimierungsproblems
Es sei das Optimierungsproblem $\Pi : \max\{f(x) \mid x \in S\}$ gegeben. Das Optimierungsproblem $\Pi' : \max\{f(x) \mid x \in S'\}$ ist eine **Relaxierung von Π** falls gilt $S \subseteq S'$.

⇨ LP Relaxierung mit ganzen Zahlen:
$\max\{c^\top x \mid Ax \leq b,\ x \in \mathbb{Z}^n\}$
wird zu
$\max\{c^\top x \mid Ax \leq b\}$

⇨ LP Relaxierung mit binären Zahlen:
$\max\{c^\top x \mid Ax \leq b,\ x \in \{0,1\}^n\}$
wird zu
$\max\{c^\top x \mid Ax \leq b,\ 0 \leq x \leq 1\}$

⇨ Optimale Zielfunktionswert der LP-Relaxierung liefert eine obere Schranke für de optimalen Zielfunktionswert des ILPs (im Falle einer Maximierung)

Knapsack Problem Beispiel

Eine Firma möchte b Franken in eine Auswahl von n Projekten investieren. Jedes Projekt i $(i = 1, \ldots, n)$ benötigt eine Investition in der Höhe a_i Franken und verspricht einen Gewinn von c_i Franken. Wie soll investiert werden, damit der Gewinn maximal wird?

Variablen:
x0..xi: Projekt i wird investiert
Zielfunktion:
sum(i=0 bis n) ci * xi -> max
Nebenbedingungen:
sum(i=0 bis n) ai * xi <= b
xi in {0,1}, für i = 0..n

Idee von B&B ist hier dass wir das ILP Teilprobleme durch LP-Relaxierung «lösen» sprich wir bekommen dann eine obere bzw. untere Schranke und durch B&B kann man dann das Optimum suchen

LP-Relaxierung:

- Binärbedingung zu 0 <= xi <= 1 lockern
- Dies erlaubt Teilinvestitionen in Projekten

Branch and Bound mit LP-Relaxierung

⇨ B&B ist eine allgemeine Methode zum Lösen von Optimierungsproblemen (nicht nur ILP)

Ablauf:

1. Lösungsraum in kleinere Mengen iterativ aufteilen (Branch)
2. In jedem Subproblem:
   a. Berechne obere Schranke (Bound) z. B. mit Relaxierung
   b. Bestimme (wenn möglich) zulässige Lösung untere Schranke (Bound)
3. Nutze diese Schrankeninfo zum Wegschneiden (Pruning)

⇨ Wenn LP-Relaxierung unzulässig dann Subproblem unzulässig => Fertig (nicht mehr weiter machen)

⇨ Wenn LP-Relaxierung ganzzahlige optimale Lösung besitzt (Blatt) => Lösungskandidat gefunden (merke Kandidaten f_delta)

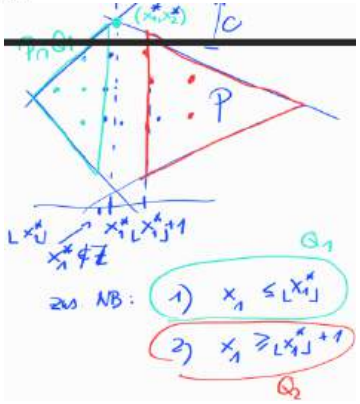
⇨ Wenn LP-Relaxierung nicht-ganzzahlige optimale Lösung besitzt (Blatt) => Obere Schranke im Knoten (obere Schranke <= Wert des letzten gemerkten Kandidaten => Knoten fertig)

⇨ Suche eine zulässige Lösung des ganzzahligen Subproblems => untere Schranke (untere Schranke = obere Schranke => Lösungskandidat gefunden)

⇨ Wenn gebrochene optimale Lösung der LP-Relaxierung im Knoten: (Branching)

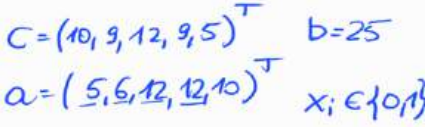
Suche eine Variable x_j, die einen nicht ganzzahligen Wert $k_j + f_j$ besitzt, wobei $k_j \in \mathbb{Z}$ und $0 < f_j < 1$. Produziere zwei neue Knoten:

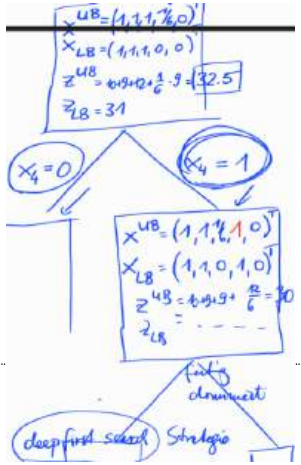
- im ersten wird die Nebenbedingung $x_j \leq k_j$ dazu genommen;
- im zweiten wird die Nebenbedingung $x_j \geq k_j + 1$ dazu genommen.



Beispiel:

⇨ Branching hier mit kj = 0 da kj + fj = 1/6





Konkretes Knapsack B&B Beispiel:

Ein Wanderer hat 4 Gegenstände mit den Gewichten 10, 7, 5, 4 kg, die er auf eine Wanderung mitnehmen möchte. Den Nutzen der Gegenstände bewertet er mit 25, 21, 30, 8. Er kann maximal 20 kg transportieren. Welche Gegenstände soll er mitnehmen, um einen möglichst grossen Nutzen zu erzielen?

Lösen Sie das Problem mit dem besprochenen Branch-and-Bound-Verfahren. Als Strategie zur Knotenwahl wählen Sie eine *Depth-First-Search* Strategie, d.h. bearbeiten Sie stets denjenigen Nachfolgeknoten zuerst, welcher durch *Aufrunden* der fraktionalen Variablen zu Stande kommt.

ILP:

Parameter:

n = (25, 21, 30, 8).T

g = (10, 7, 5, 4).T

Variablen:

xi: i in 0..3 und Gegenstand i wird mitgenommen falls xi = 1

Zielfunktion:

sum(i=0, i=3) ni * xi -> max (n*x->max)

Nebenbedingungen:

sum(i=0, i=3) gi * xi <= 20 (g*x <= 20)

xi = {0, 1}

LP-Relaxierung:

0 <= xi <= 1

1. **Initialisierung:**
   - Setze die beste gefundene Lösung $x^* = \text{null}$ (oder einen sehr schlechten Wert)
   - Erstelle eine Liste (Queue/Stack) mit dem Startproblem (ganzer Lösungsraum)
2. **Solange es ungelöste Teilprobleme gibt:**
   a. **Wähle ein Teilproblem aus der Liste**
   b. **LP-Relaxierung lösen → Lösung x^{LP} mit Zielfunktionswert z^{LP}**
   c. **Falls LP-Relaxierung unzulässig:**
   - Verwerfe das Teilproblem (kein weiteres Vorgehen)

   d. **Sonst, falls $z^{LP} \leq$ bester bisher gefundener Zielfunktionswert:**
   - Verwerfe das Teilproblem (Pruning)

   e. **Sonst, wenn x^{LP} ganzzahlig:**
   - Aktualisiere beste Lösung $x^* \leftarrow x^{LP}$ (falls besser)
   - Verwerfe das Teilproblem (abgeschlossen)

   f. **Sonst (LP-Lösung nicht ganzzahlig):**
   - Wähle eine Variable x_j mit nicht-ganzzahligem Wert x_j^{LP}
   - Erzeuge zwei Teilprobleme mit Nebenbedingungen:
     - $x_j \leq \lfloor x_j^{LP} \rfloor$
     - $x_j \geq \lceil x_j^{LP} \rceil$
   - Füge beide Teilprobleme zur Liste hinzu
3. **Ende:**
   - Die beste gespeicherte Lösung x^* ist optimal (oder keine Lösung gefunden)

Traveling Salesmen Problem als ILP

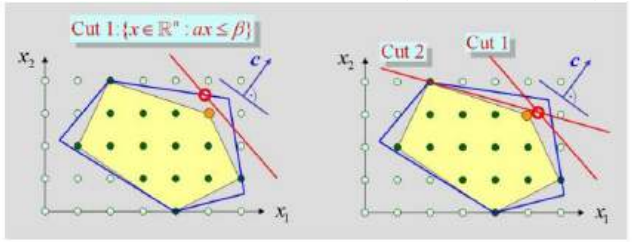
Definition

Algorithmisch:

- ⇨ Komplette Enumeration = (n-1)!/2 ((n-1)! Wegen Permutation der übrigen Städte (ausser Start) und / 2 wegen jeder Rundreise ist gleich ihrer Umkehrung)
- ⇨ mit DP n*2^n
- ⇨ mit B&B variabel meist exponentiell
- ⇨ Oder andere Verfahren die verbessernd oder konstruktiv sind, aber keine optimale Lösung finden

Exakt:

- ⇨ Kombination von Branch & Bound und Schnittebenen Verfahren (Branch and Cut)



ILP Formulierung 1 (intuitiv)

Parameter:
D (Distanzmatrix) => dij = Distanz von Stadt i nach Stadt j
Variable:
xij = 1 wenn Stadt i nach Stadt j in Tour, 0 sonst
Zielfunktion:
sum(i=0 bis n) sum(j=0 bis n mit j!=i) dij*xij -> min
Nebenbedingungen:
Jede Zeile und Spalte nur eine 1 (Jede Stadt genau einmal betreten und verlassen)
sum(i=0 bis n) xij = 1, für j = 0..n
sum(j=0 bis n) xij = 1, für i = 0..n
Keine Subtouren (eine einzelne Rundtour)
…
Ausführlich:

$$\sum_{i \in V} \sum_{j \in V \setminus \{i\}} d_{ij} x_{ij} \rightarrow \min \quad (1)$$

$$\sum_{j \in V \setminus \{i\}} x_{ij} = 1, \quad i \in V \quad (2)$$

$$\sum_{i \in V \setminus \{j\}} x_{ij} = 1, \quad j \in V \quad (3)$$

$$\sum_{i \in S} \sum_{j \in S \setminus \{i\}} x_{ij} \le |S| - 1, \quad S \subset V, |S| \ge 2 \quad (4)$$

$$x_{ij} \in \{0,1\}, \quad i,j \in V, i \neq j \quad (5)$$

- von Dantzig, Fulkerson und Johnson (1954)
- Distanzen zwischen Städten gegeben durch $d_{ij}, i,j \in V$.
- Variablen x_{ij} für $i,j \in V, i \neq j$:

$$x_{ij} = \begin{cases} 1 & \text{unmittelbar nach Stadt } i \text{ kommt Stadt } j \\ 0 & \text{sonst} \end{cases}$$

- (2) sichert, dass der "Nachfolger" der Stadt i eindeutig ist
- (3) sichert, dass der "Vorgänger" der Stadt j eindeutig ist
- (4) sichert, dass keine Subtouren existieren
- Anzahl der Subtourbedingungen (4) wächst exponentiell mit Anzahl Knoten $|V|$

⇨ Wie genau werden dadurch Subtouren verhindert? Für jede echte Teilmenge S von V (Menge aller Knoten) die mindestens 2 Elemente besitzt bedeutet: Eine Subtour ist ein geschlossener Zyklus innerhalb einer Teilmenge S und eine vollständige Rundtour über S enthält genau |S| Kanten (eine von jeder Stadt zur nächsten), aber das wäre ja nicht zulässig weil wir keine Subtour wollen => Deshalb höchstens |S| - 1 Kanten

ILP Formulierung 2

⇨ Problem mit Formulierung 1: Es gibt exponentiell viele Subtouren Teilmengen

$$\sum_{i=1}^{n} \sum_{j=1, j \neq i}^{n} d_{ij} x_{ij} \rightarrow \min \quad (6)$$

$$\sum_{j=1, j \neq i}^{n} x_{ij} = 1, \quad i = 1, \ldots, n \quad (7)$$

$$\sum_{i=1, i \neq j}^{n} x_{ij} = 1, \quad j = 1, \ldots, n \quad (8)$$

$$u_i - u_j + n x_{ij} \le n-1, \quad 2 \le i \neq j \le n \quad (9)$$

$$x_{ij} \in \{0,1\}, \quad i,j = 1, \ldots, n, i \neq j \quad (10)$$

⇨ Subtourenbedingungen (9) sichern, dass jede Subtour den Knoten 1 beinhalten muss. (Braucht aber mehr Vars)
⇨ Subtouren werden durch Hilfsvariablen ui verhindert, die einer Art Position in der Tour (für jede Stadt i) kodieren.
⇨ Wenn xij = 1 (Reise von i nach j) dann muss gelten uj >= ui + 1 (Stadt j kommt nach Stadt i) => u speichert Pos von i und somit kann keine permu. entstehen
⇨ Anzahl der Subtourenbedingungen wächst somit nur noch quadratisch

Widerspruch erzeugt bei Subtour:
Annahme: Subtour mit 3 Städten {1,2,3} => x12 = 1, x23 = 1, x31 = 1
Dann Bedingungen: u2 >= u1 + 1, u3 >= u2 + 1, u1 >= u3 + 1
⇨ Kette der Ungleichungen: u1 >= u1 + 3 => 0 >= 3 => Widerspruch

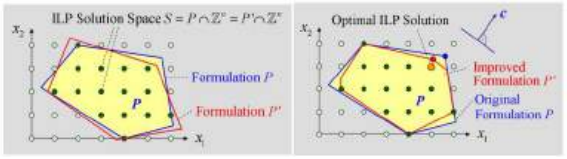
Allgmeiner Recap:
Der optimale Punkt liegt auf einer äusseren Ecke der Polyeders P
Die Richtung wird durch den Zielfunktionsvektor c bestimmt (Normalenvektor auf Zielfunktions-Hyperebene da c.T*x = z).
Bester Punkt ist der letzte der mit Verschiebung von Zielfunktion in Richtung c kontakt hat (y-Achsenabschnitt bspw. maximierend)

Formulierungen von ILPs

Idee

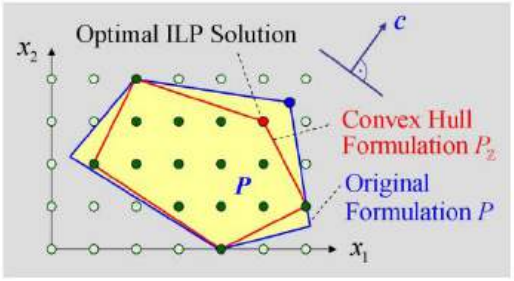
$\max\{\boldsymbol{c}^\top \boldsymbol{x} \mid \boldsymbol{x} \in P \cap \mathbb{Z}^n\}$ mit $P = \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}\}$

- ⇨ P (Polyeder) ist die Menge der zulässigen Lösungen, die die Nebenbedingungen (Ungleichungen) erfüllen und danach von der Zielfunktion hier maximiert wird, um das x in P zu finden (Zielfunktion über den Vektor c) bei dem die Zielfunktion maximiert wird
- ⇨ Jedes ILP hat unendlich viele Formulierungen
- ⇨ Jedes ILP kann auch genau x Lösungen haben (anders als ein LP wo bei mehr als 2 optimalen Punkten unendlich viele existieren)
- ⇨ Falls $P \cap \mathbb{Z}^n = P' \cap \mathbb{Z}^n$ dann ergibt sich das gleiche ILP
- ⇨ Fall zusätzlich $P' \subseteq P$ erhalten wir eine besser Formulierung: LP-Relaxierung ergibt die gleiche oder bessere Schranke
- ⇨ Für ganzzahlige Lösungen sind P und P' somit gleich aber für nicht ganzzahlige ist P' enger, d.h. LP-Relaxierung kann besser gewählte werden



Beste Formulierung

- ⇨ Für jedes ILP existiert (unter bestimmten Voraussetzungen) eine beste Formulierung



- ⇨ Es gibt Problem (wie Transportproblem, spezielle Struktur, spezielle Matrixeigenschaften) bei denen die beste Formulierung bekannt ist
- ⇨ Schwierig das zu finden (im Allgemeinen).
- ⇨ Man kann versuchen, die mathematische polyedrische Struktur des Problems besser zu verstehen um punktuell bessere Beschreibungen des Problems herzuleiten

Losgrössenplannung Beispiel

Eine Firma möchte ihre Produktion für die nächsten N Tage planen. Falls an einem Tag produziert wird, fallen Setup-Kosten der Grösse r an. Der Abnehmerbedarf am Tag t ist durch d_t gegeben $(t = 1, \ldots, N)$. Bereits produzierte Ware, die noch nicht geliefert wurde, verursacht Lagerkosten in Höhe f pro Einheit und Tag. Im Moment ist das Lager leer.

Wie soll die Firma ihre Produktion planen, damit die Kosten minimal werden?

Einfaches formuliertes Modell:

Mengen:

Zeithorizont $T = \{1, \ldots, N\}$

Parameter:

d_t: Bedarf in Periode $t \in T$
r: Setup-Kosten pro Los
f: Lagerkosten pro Einheit und Periode

Variablen:

x_t: Produktion in Periode $t \in T$
y_t: Lagerbestand am Ende von Periode $t \in T$
z_t: Binärer Indikator, der angibt, ob in Periode $t \in T$ produziert wird

- ⇨ Entscheidungsvariablen immer dort verwenden, wo man etwas beeinflussen kann: Wie hier sicher die Anzahl der Produktion an einem Tag, aber auch wie
- ⇨ Da man nicht if x==0 then keine setup kosten machen kann braucht man eine Hilfsvariable für binären Indikator

d_t: Bedarf in Periode $t \in T$
r: Setup-Kosten pro Los
f: Lagerkosten pro Einheit und Periode
M: "big M", grösser als maximal mögliche Losgrösse

setup Lager

$$
\begin{aligned}
r\sum_{t\in T} z_t + f\sum_{t\in T} y_t &\rightarrow \min \\
x_1 - d_1 &= y_1 \\
y_{t-1} + x_t - d_t &= y_t \qquad t \in T \setminus \{1\} \\
x_t &\leq M z_t \qquad t \in T \\
x_t, y_t &\geq 0 \qquad t \in T \\
z_t &\in \{0,1\} \qquad t \in T
\end{aligned}
$$

- ⇨ Wenn nicht produziert wird, muss xt 0 sein (M * 0 = 0) und wenn produziert wird dafür irgendeine Zahl, die aber kleiner ist als maximal mögliche Losgrösse => x kann nicht grösser als M sein)
- ⇨ Es wäre auch möglich ohne y, indem mach jeweils immer anstatt yt-1 einfach Summe von 0 bis xt-1 – Summe von 0 bis dt-1 rechnet

⇨ Einfaches Modell weiss nicht, wofür Lagerbestand gedacht ist. Es wird einfach nur auf Halde produziert

Besser formuliertes Modell:

Variablen:

x_{it}: Produktion in Periode $i \in T$ zum Verbrauch in Periode $t \in T$ bestimmt

y_{it}: Lagerbestand am Ende von Periode $i \in T$, der zum Verbrauch in Periode $t \in T$ bestimmt ist

z_i: Binärer Indikator, der angibt, ob in Periode $i \in T$ produziert wird

$$
\begin{aligned}
r\sum_{i\in T} z_i + f\sum_{i\in T}\sum_{t\in T} y_{it} &\rightarrow \min \\
x_{11} - d_1 &= 0 \\
x_{1t} &= y_{1t} && t \in T\setminus\{1\} \\
y_{t-1,t} + x_{tt} - d_t &= 0 && t \in T\setminus\{1\} \\
y_{i-1,t} + x_{it} &= y_{it} && i \in T\setminus\{1\}, t \in T, i < t \\
x_{it} &\leq M_t z_i && i,t \in T, i \leq t \\
x_{it} &= 0 && i,t \in T, i > t \\
y_{it} &= 0 && i,t \in T, i \geq t \\
x_{it}, y_{it} &\geq 0 && i,t \in T \\
z_i &\in \{0,1\} && i \in T
\end{aligned}
$$

⇨ Wir berechnen nun für jeden Tag die Menge zusätzlich, für dann für eine Periode t in der Zukunft verwendet wird. (D.h. jede Periode t hat t Zeilen, um die Lagerbestände von diesem Tag aufzuteilen), Somit entsteht eine obere 3ecks Matrix (unten immer 0 da man ja nichts für gestern lagern kann)

⇨ Das ist eine bessere Formulierung, da neben den von der einfachen Modellierung genauere Angaben gemacht werden (wie oben P')

⇨ Das erweiterte Modell macht deine Problembeschreibung genauer und strenger, so dass du beim «lockeren» Durchprobieren (Relaxierung) weniger Unsinn bekommst und somit das ILP effizienter gelöst wird, auch wenn das Modell grösser ist.

Logische Nebenbedingungen

Binäre Variablen x, y:

- $x = 1$ oder $y = 1$: $x + y \geq 1$
- $x = 1 \Rightarrow y = 1$: $x \leq y$

Beliebige Variablen $x_1, \ldots, x_n$:

- $f(x_1, \ldots, x_n) \leq 0$ oder $g(x_1, \ldots, x_n) \leq 0$ wird umformuliert zu

$$
\begin{aligned}
f(x_1, \ldots, x_n) &\leq My \\
g(x_1, \ldots, x_n) &\leq M(1-y)
\end{aligned}
$$

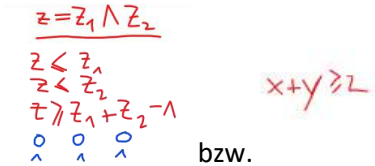
wobei y eine binäre Variable ist und M eine "genügend grosse" Konstante.

- $f(x_1, \ldots, x_n) < 0 \Rightarrow g(x_1, \ldots, x_n) \leq 0$ wird umformuliert zu $-f(x_1, \ldots, x_n) \leq 0$ oder $g(x_1, \ldots, x_n) \leq 0$

Übersicht:

⇨ Logische Bedingungen mit binären Vars:

AND:



bzw.

OR:

bzw.

Autohersteller Beispiel

Ein Autohersteller möchte drei Autotypen produzieren: kompakt, mittel und gross. Die benötigten Ressourcen und der jeweilige Gewinn pro Auto sind in der folgenden Tabelle angegeben:

Ressource	Autotyp Kompakt	Mittel	Gross
Stahl (t)	1.5	3	5
Arbeit (h)	30	25	40
Profit (Fr.)	2000	3000	4000

Zur Verfügung stehen 6000 Tonnen Stahl und 60'000 Arbeitsstunden. Aus betrieblichen Gründen müssen entweder mindestens 1'000 Stück von einem Autotyp hergestellt werden, oder der Autotyp wird gar nicht hergestellt. Wie soll der Autohersteller seine Produktion planen?

Variablen:

xk,xm,xg: Mengen von Autotyp

zk,zm,zg: Binäre Variablen für wird hergestellt oder nicht

Zielfunktion:

2000 * xk + 3000 * xm + 4000 * xg -> max

Nebenbedingungen:

1.5 * xk + 3 * xm + 5 * xg <= 6000

30 * xk + 25 * xm + 40 * xg <= 30000

Mindestens 1000 oder gar nicht:

M = Die grösste mögliche Stückzahl (6000 / 1.5)

xk <= M * zk

xk >= 1000 * zk

(Für m und k gleich)

Oder alternativ: (Branching: x1 <= 0 oder x1>=1000 zu x1 <= 0 oder 1000 – x1 <= 0 schreiben können (beide rechte Seiten = 0))

xk <= M * zk

1000 – xk <= M * (1 – zk) => wenn zk = 0 dann x1 >= 1000

xk,xm,xg >= 0 und Ganzzahlig => Natürliche Zahlen