# Simplex und das Branch-and-Bound-Verfahren mit Implementierung in Python

## Bachelorarbeit

vorgelegt von

Christian Jäkle

an der


Mathematisch-Naturwissenschaftliche Sektion
Fachbereich Mathematik und Statistik

Gutachter: Herr Prof. Dr. Stefan Volkwein

Konstanz, 28. Oktober 2017

## Selbstständigkeitserklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit mit dem Thema

Simplex und das Branch-and-Bound-Verfahren mit Implementierung in Python

selbständig verfasst und keine anderen Hilfsmittel als die angegebenen benutzt habe. Die Stellen, die anderen Werken dem Wortlaut oder dem Sinne nach entnommen sind, habe ich in jedem einzelnen Falle durch Angaben der Quelle, auch der benutzten Sekundärliteratur, als Entlehnung kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Konstanz, 28. Oktober 2017

_____

Christian Jäkle

# Inhaltsverzeichnis

# 1 Einleitung

In dieser Arbeit wollen wir eine sowohl theoretische wie praktische Einführung in die Lineare Programmierung geben. Dabei werden wir zunächst mit etwas Theorie zu linearen Programmen sowie zu Dualität beginnen. Im Anschluss wird das Simplex-Verfahren vorgestellt, das als eines der Standardverfahren für Lineare Programmierung gilt. Nach dieser Einführung werden wir Lineare Programme betrachten, bei denen die zusätzliche Restriktion auftritt, dass einige oder alle gesuchten Größen ganzzahlig sein müssen. Für solche Problemstellungen stellen wir das Branch-and-Bound-Verfahren vor.

Im praktischen Teil dieser Arbeit werden wir die beiden Verfahren in Python implementieren und einen Vergleich mit dem kommerziellen Programm Matlab aufstellen. Im Anhang sind die vollständigen Codes der jeweiligen Implementierungen zu finden.

# 2 Simplex

## 2.1 Einführende Bemerkungen

In diesem Kapitel werden wir einige Definitionen und Sätze aus der Numerik wiederholen. Die Beweise dieser sind gänzlich im dritten Kapitel von [Lu15] zu finden.

### 2.1.1 Lineare Optimierungsprobleme

**Definition 2.1.1** Ein lineares Optimierungsproblem (LP) ist gegeben durch:

$$\begin{aligned} &\max \sum_{j=1}^{n} c_j x_j \\ \text{u.d.N.} &\sum_{j=1}^{n} a_{ij} x_j \quad \leq b_i, \qquad i = 1, \ldots, m \\ &\qquad x_j \quad \geq 0, \qquad j = 1, \ldots, n \end{aligned} \tag{2.1.1}$$

bzw. max $c^T x$ u.d.N. $Px \leq b$, $x \geq 0$ mit $c \in \mathbb{R}^n$, $P \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$ und eine zu suchende Größe $x \in \mathbb{R}^n$. Wir nennen (2.1.1) auch Normalform I und die zu maximierende Funktion Zielfunktion.
Mit der Einführung sogenannter Schlupfvariablen $s = (s_1, \ldots, s_m) \in \mathbb{R}^m$, die die Ungleichungen einfach zu Gleichungen machen, erhalten wir mit $A \in \mathbb{R}^{m \times (m+n)}$,
$rk(A) = m$, $b \in \mathbb{R}^m$ und $d \in \mathbb{R}^{m+n}$

$$\begin{aligned} &max \quad d^T x \\ u.d.N. \quad &Az \quad = b \\ &z \quad \geq 0 \end{aligned} \tag{2.1.2}$$

wobei $A = [P\, I]$, $z = (x, s)^T$ und $d = (c, 0)^T$ und nennen (2.1.2) Normalform II. Hier ist die zu suchende Größe $z \in \mathbb{R}^{m+n}$.

**Bemerkung 2.1.2** Jedes LP lässt sich in eines der obigen transformieren z.B.

1. Ist min $c^T x$ äquivalent zu max $(-c^T x)$ bzw. es gilt min $c^T x = -\max(-c^T x)$.
2. Eine Nebenbedingung der Form $a_{k1} x_1 + \cdots + a_{kn} x_n \geq b_i$ wird zu $-a_{k1} x_1 - \cdots - a_{kn} x_n \leq -b_i$
3. Falls ein $x_k$ keiner Vorzeichenbeschränkung unterliegt, so setzt man $x_k = x_{k1} - x_{k2}$, $x_{k1} \geq 0$, $x_{k2} \geq 0$.

Wir formulieren jetzt noch einige Eigenschaften des zulässigen Bereichs

$$\begin{aligned} K &:= \{x \in \mathbb{R}^n : x \geq 0, Px \leq b\} \qquad \text{für LPs in Normalform I} \\ K &:= \{z \in \mathbb{R}^{m+n} : z \geq 0, Az = b\} \qquad \text{für LPs in Normalform II} \end{aligned}$$

**Definition 2.1.3**

1. Eine Teilmenge $M \subseteq \mathbb{R}^k$ heißt konvex, wenn für alle $x, y \in M$ und alle $0 \leq \lambda \leq 1$ gilt: $z := \lambda x + (1-\lambda)y \in M$.
2. Seien $M$ konvex, $x_1, \ldots, x_r \in M, \mu_1, \ldots, \mu_r \in \mathbb{R}, \mu_i \geq 0$ und $\sum_{i=1}^r \mu_i = 1$. Dann heißt $x = \sum_{i=1}^r \mu_i x_i \in M$ konvexe Linearkombination.
3. Sei $M$ konvex. Ein $x \in M$ heißt Ecke, wenn aus $x = \mu x_1 + (1-\mu)x_2$, $0 < \mu < 1$, $x_1, x_2 \in M$ stets $x = x_1 = x_2$ folgt (d.h. $x$ kann nicht als echte konvexe Linearkombination dargestellt werden).

Man sieht sofort, dass der Durchschnitt beliebig vieler konvexer Mengen konvex ist.

**Bemerkung 2.1.4** Eine beliebige konvexe Menge kann unendlich viele Ecken haben. So ist zum Beispiel jeder Kreis konvex und jeder Randpunkt ist eine Ecke.

**Definition 2.1.5** Seien $\alpha \in \mathbb{R}^k \backslash \{0\}$ *und* $\gamma \in \mathbb{R}$. Die Menge

$$H := \{x \in \mathbb{R}^k : \alpha^T x = \gamma\}$$

heißt Hyperebene.

Eine Hyperebene teilt den $\mathbb{R}^k$ in die beiden Halbräume

$$H^+ := \{x \in \mathbb{R}^k : \alpha^T x \geq \gamma\} \qquad \text{und} \qquad H^- := \{x \in \mathbb{R}^k : \alpha^T x \leq \gamma\}$$

Man kann zeigen, dass $H, H^+$ und $H^-$ abgeschlossene, konvexe Mengen sind. Nebenbedingungen der Form $p_{i1}x_1 + \cdots + p_{in}x_n \leq b_i$ bzw. $x_i \geq 0$ liefern also abgeschlossene Halbräume. Nebenbedingungen der Form $a_{i1}z_1 + \cdots + a_{in}z_n = b_i$ ergeben abgeschlossene Hyperebenen. Damit erhalten wir, dass der zulässige Bereich $K$ ein Durchschnitt von endlich vielen abgeschlossenen Hyperebenen und Halbräumen ist.

**Satz 2.1.6** Der zulässige Bereich $K$ ist konvex und abgeschlossen.

**Korollar 2.1.7** Ist $K \neq \emptyset$ und beschränkt, so hat das LP (2.1.1) eine eindeutige Lösung.

*Beweis:* Die Zielfunktion ist als lineare Funktion stetig, und $K$ ist als beschränkte und abgeschlossene Teilmenge des $\mathbb{R}^k$ kompakt. Da stetige Funktionen auf kompakten Mengen ihre Extremwerte annehmen, folgt bereits die Behauptung. □

### 2.1.2 Ecken des zulässigen Bereichs

Gegeben sei ein LP in Normalform II wie in (2.1.2). Seien $A = (a_1, \ldots, a_{m+n})$ mit $a_i \in \mathbb{R}^m$ (aufgefasst als Spalten von $A$) und $K$ der entsprechende zulässige Bereich. Für $x \in K$ bezeichne $I(x) := \{j \in \{1, \ldots, m+n\} : x_j > 0\}$. Man kann zeigen:

- Für $x \in K$ sind äquivalent:
  1. $x$ ist eine Ecke von $K$
  2. Die Vektoren $a_j$, $j \in I(x)$ sind linear unabhängig.
- Der zulässige Bereich $K \subset \mathbb{R}^{(m+n)}, K \neq \emptyset$ besitzt Ecken.

**Definition 2.1.8** Seien $A \in \mathbb{R}^{m\times(m+n)}, rk(A) = m$, $B = (a_{i_1}, \ldots, a_{i_m})$ eine Teilmatrix von $A$ mit $rk(B) = m$. Ein $x \in \mathbb{R}^{m+n}, x \geq 0$, heißt Basispunkt zu $B$, falls

$$x_j = 0 \,\text{für}\, j \notin \{i_1, \ldots, i_m\} \,\text{und}\, \sum_{j=1}^{m} x_{i_j} a_{i_j} = b$$

gelten. Die Komponenten $x_{i_1}, \ldots, x_{i_m}$ werden als Basisvariablen bezeichnet. Der Punkt $x \in \mathbb{R}^{m+m}$ heißt Basispunkt/Basislösung, wenn es eine Teilmatrix $B$ von $A$ gibt, sodass $x$ ein Basispunkt von $B$ ist.

Man kann folgende Sätze zeigen:

**Satz 2.1.9** Sei $rk(A) = m$ und $x \in \mathbb{R}^{m+n}$. Dann sind äquivalent:

1. $x$ ist eine Ecke von $K$
2. $x$ ist eine Basislösung

**Korollar 2.1.10** Der zulässige Bereich $K$ besitzt höchstens $\binom{m+n}{m}$ Ecken.

**Satz 2.1.11** Der zulässige Bereich $K$ sei beschränkt und nicht leer. Dann nimmt die Zielfunktion $c^T x$ in (2.1.2) ihr Maximum in einer Ecke an.

## 2.2 Dualität

Wir wollen zunächst festhalten, dass eine Zielfunktion unter gewissen Umständen beliebig vergrößert werden kann, z.B. wenn der zulässige Bereich unbeschränkt ist. Wir nennen solch ein LP unbeschränkt. Wie wir erkennen, wann ein LP unbeschränkt ist und damit keine Lösung besitzt, werden wir später sehen. Die Beweise der folgenden Sätze sind unter anderem im fünften Kapitel von [Van14] zu finden.

**Definition 2.2.1** Gegeben sei ein primales LP mit

$$\begin{aligned} \max \quad & c^T x \\ \text{u.d.N.} \quad & Ax \leq b, \\ & x \geq 0, \end{aligned} \tag{2.2.1}$$

wobei $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m\times n}$ und $b \in \mathbb{R}^n$. Dann definieren wir das duale LP zu (2.2.1):

$$\begin{aligned} \min \quad & b^T y \\ \text{u.d.N.} \quad & A^T y \geq c, \\ & y \geq 0. \end{aligned} \tag{2.2.2}$$

Wir sehen sofort, dass das duale LP zu (2.2.2) wieder (2.2.1) ist (beachte, dass gilt: $\max f = -\min -f$). Im Folgenden formulieren wir die beiden Dualitätssätze und die Komplementaritätsbedingungen. Diese geben uns hinreichende und notwendige Optimalitätsbedingungen.

**Satz 2.2.2** (Schwacher Dualitätssatz) Seien $x \in \mathbb{R}^n$ zulässig für das primale LP (2.2.1) und $y \in \mathbb{R}^m$ zulässig für das duale LP (2.2.2). Dann gilt

$$c^T x \leq b^T y$$

*Beweis:* Es gilt

$$c^T x \leq (A^T y)^T x = y^T A x \leq y^T b, \tag{2.2.3}$$

wobei die erste Ungleichung aus den Nebenbedingungen von (2.2.2) folgt, die zweite Ungleichung aus den Nebenbedingungen von (2.2.1). □

**Korollar 2.2.3** Seien $x^*$ eine primale, zulässige Lösung von (2.2.1) und $y^*$ eine duale, zulässige Lösung von (2.2.2), sodass

$$c^T x^* = b^T y^*$$

gilt. Dann sind beide Lösungen für ihr jeweiliges Problem optimal.

Weiter kann man folgende Sätze zeigen: [1]

**Satz 2.2.4** (Starker Dualitätssatz)

- Hat eines der beiden LPs ((2.2.1) oder (2.2.2)) eine endliche Optimallösung, so auch das andere, und die optimalen Zielfunktionswerte sind gleich.
- Ist eines der LPs unbeschränkt, so ist das dazu duale Problem unzulässig.

**Satz 2.2.5** (Komplementaritätsbedingungen) Seien $x = (x_1, \ldots, x_n)$ zulässig für (2.2.1) und $y = (y_1, \ldots, y_m)$ zulässig für (2.2.2). Seien weiter $(w_1, \ldots, w_m)$ die zugehörigen primalen Schlupfvariablen und $(z_1, \ldots, z_n)$ die zugehörigen dualen Schlupfvariablen. Dann sind $x$ und $y$ optimal für ihr jeweiliges Problem genau dann, wenn

$$\begin{aligned} x_j z_j &= 0 \qquad \text{für } j = 1, \ldots, n \\ w_i y_i &= 0 \qquad \text{für } i = 1, \ldots, m \end{aligned}$$

gelten.

## 2.3 Der Simplex Algorithmus

Die Idee des Simplex Algorithmus beruht darauf, dass in jeder Iteration der Simplex Algorithmus die Ecken entlangwandert und dabei versucht, den Wert der Zielfunktion zu vergrößern. Da es nur eine endliche Anzahl an Ecken gibt, sollte es so sein, dass der Algorithmus mit der gewünschten Information terminiert, was in den meisten Fällen auch der Fall ist. Der hier gewählte Aufbau ist in [Van14], Kapitel 6, zu finden.

[1] Sehr ausführliche Beweise der Sätze 2.2.4 und 2.2.5 sind in [Van14], Kapitel 5.4 und 5.5, zu finden. Hier werden die Sätze aber für LPs in Normalform I gezeigt. Sehr kurze Beweise sind in [Ham16], Kapitel 3.1 und 3.2, zu finden. Hier werden sie für Probleme in Normalform II gezeigt.

### 2.3.1 LPs in Matrix-Notation

Gegeben sei ein LP in Normalform I:

$$\begin{aligned} &\max \sum_{j=1}^{n} c_j x_j \\ \text{u.d.N.} &\sum_{j=1}^{n} a_{ij} x_j \quad \leq b_i \qquad i = 1, \ldots, m \\ &x_j \quad \geq 0 \qquad j = 1, \ldots, n. \end{aligned} \tag{2.3.1}$$

Wir führen Schlupfvariablen $x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij} x_j$, $i = 1, \ldots, m$ ein und können damit das Problem in Normaform II wie folgt formulieren:

$$\begin{aligned} \max \quad & c^T x \\ \text{u.d.N.} \quad & Ax = b, \\ & x \geq 0 \quad \text{mit} \end{aligned} \tag{2.3.2}$$

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} & 1 & & \\ \vdots & & \vdots & & \ddots & \\ a_{m1} & \cdots & a_{mn} & & & 1 \end{pmatrix},$$

$$b = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}, \quad c = \begin{pmatrix} c_1 \\ \vdots \\ c_n \\ 0 \\ \vdots \\ 0 \end{pmatrix} \quad \text{und} \quad x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \\ x_{n+1} \\ \vdots \\ x_{n+m} \end{pmatrix}. \tag{2.3.3}$$

Setzen wir nun $x_1 = \cdots = x_n = 0$ und $x_{n+i} = b_i$, für $i = 1 \ldots, m$ so erhalten wir eine zulässige Lösung für (2.3.1) (sofern die $b_i \geq 0$, der andere Fall wird später behandelt). Aus Kapitel (2.1) wissen wir, dass wir in einer Ecke des zulässigen Bereichs sind und unsere $x_{n+1}, \ldots, x_{n+m}$ die (linear unabhängigen) Basisvariablen sind. Wir nennen die übrigen $x_i$ die Nicht-Basisvariablen und bezeichnen mit $\mathcal{B}$ die Menge, die die Indices der Basisvariablen enthält und mit $\mathcal{N}$ die Menge, die die Indices der Nicht-Basisvariablen enthält. In jeder Iteration des Algorithmus wird genau eine Basisvariable zu einer Nicht-Basisvariable und umgekehrt. Wir wollen mit $f = c^T x$ den Wert der Zielfunktion bezeichnen.
Sei $A = [B\, N]$, wobei $B \in \mathbb{R}^{m \times m}$ die Matrix ist, die aus den $m$ Spalten von $A$ entsteht, die die Basisvariablen repräsentieren und $N \in \mathbb{R}^{m \times n}$ die, die die $n$ Spalten von $A$ der Nicht-Basisvariblen repräsentiert. Streng genommen gilt nicht $A = [B\, N]$, sondern erst nach passenden Zeilenpermutationen. Hier wie im folgenden, nehmen wir die Gleichheit aber ohne Einschränkung an. Das gleiche machen wir mit $x$ und $c$, also

$$x = \begin{bmatrix} x_{\mathcal{B}} \\ x_{\mathcal{N}} \end{bmatrix}, \qquad c = \begin{bmatrix} c_{\mathcal{B}} \\ c_{\mathcal{N}} \end{bmatrix}.$$

Dann gilt:

$$Ax = [B\,N]\begin{bmatrix} x_{\mathcal{B}} \\ x_{\mathcal{N}} \end{bmatrix} = Bx_{\mathcal{B}} + Nx_{\mathcal{N}} \qquad \text{und} \quad c^T x = \begin{bmatrix} c_{\mathcal{B}} & c_{\mathcal{N}} \end{bmatrix}\begin{bmatrix} x_{\mathcal{B}} \\ x_{\mathcal{N}} \end{bmatrix} = c_{\mathcal{B}}^T x_{\mathcal{B}} + c_{\mathcal{N}}^T x_{\mathcal{N}}.$$

Wir nennen dann den Wert unserer Zielfunktion $f$ und die Werte aller Variablen $x_i$ Dictionary. Analog notieren wir dies für das duale Problem.

### 2.3.2 Primaler Simplex-Algorithmus

Aus $Ax = b$, was $Bx_{\mathcal{B}}+Nx_{\mathcal{N}} = b$ impliziert, und der Tatsache, dass unsere Basisvariablen linear unabhängig sind (vgl. Kapitel (2.1)), erhalten wir, dass $B$ invertierbar ist. Damit erhalten wir:

$$x_{\mathcal{B}} = B^{-1}b - B^{-1}Nx_{\mathcal{N}}. \qquad (1)$$

Setzen wir dies für den Wert $f$ der Zielfunktion ein, erhalten wir:

$$\left.\begin{aligned} f &= c_{\mathcal{B}}^T x_{\mathcal{B}} + c_{\mathcal{N}}^T x_{\mathcal{N}} \\ &= c_{\mathcal{B}}^T(B^{-1}b - B^{-1}Nx_{\mathcal{N}}) + c_{\mathcal{N}}^T x_{\mathcal{N}} \\ &= c_{\mathcal{B}}^T B^{-1}b - ((B^{-1}N)^T c_{\mathcal{B}} - c_{\mathcal{N}})^T x_{\mathcal{N}} \end{aligned}\right\} \qquad (2)$$

Aus (1) und (2) folgt, dass unser Dictionary auch als

$$\left.\begin{aligned} f &= c_{\mathcal{B}}^T B^{-1}b - ((B^{-1}N)^T c_{\mathcal{B}} - c_{\mathcal{N}})^T x_{\mathcal{N}} \\ x_{\mathcal{B}} &= B^{-1}b - B^{-1}Nx_{\mathcal{N}} \end{aligned}\right\} \qquad (2.3.4)$$

geschrieben werden kann.
Die Standardlösung für (2.3.4) ist $x_{\mathcal{N}}^* = 0$ und $x_{\mathcal{B}}^* = B^{-1}b$ (4)
Das Duale Problem zu (2.3.2) hat ebenfalls Schlupfvariablen. Seien diese $(z_1, \ldots, z_n)$ und die gesuchten $(y_1, \ldots, y_m)$ setzen wir zu $(z_{n+1}, \ldots, z_{n+m})$. Dann ist

$$z = (z_{n+1}, \ldots, z_{n+m})^T = [z_{\mathcal{N}},\ z_{\mathcal{B}}]^T.$$

Damit kann das duale Dictionary zu (2.3.4) als

$$\begin{aligned} -g &= -c_{\mathcal{B}}^T B^{-1}b - (B^{-1}b)^T z_{\mathcal{B}} \\ z_{\mathcal{N}} &= (B^{-1}N)^T c_{\mathcal{B}} - c_{\mathcal{N}} + (B^{-1}N)^T z_{\mathcal{B}} \end{aligned}$$

geschrieben werden, wobei mit $g$ der Wert der Zielfunktion des dualen Problems bezeichnet wird. Die duale Lösung zu diesem Dictionary ist

$$z_{\mathcal{B}}^* = 0 \text{ und } z_{\mathcal{N}} = (B^{-1}N)^T c_{\mathcal{B}} - c_{\mathcal{N}}. \qquad (5)$$

Aus (4) und (5) folgt $f^* = c_{\mathcal{B}}^T B^{-1}b$ und wir sehen, dass das primale Dictionary

$$\begin{aligned} f &= f^* - (z_{\mathcal{N}}^*)^T x_{\mathcal{N}} \\ x_{\mathcal{B}} &= x_{\mathcal{B}}^* - B^{-1}Nx_{\mathcal{N}} \end{aligned}$$

ist. Das duale Dictionary davon ist gegeben durch

$$-g = -f^* - (x^*_{\mathcal{B}})^T z_{\mathcal{B}}$$
$$z_{\mathcal{N}} = z^*_{\mathcal{N}} + (B^{-1}N)^T z_{\mathcal{B}}.$$

Eine Iteration des primalen Simplex-Algorithmus wird nun wie folgt formuliert (dabei muss $x$ natürlich primal zulässig sein, d.h $x_i \geq 0$ f. a. $i = 1, \dots m+n$):

1. Schritt: Ist die Lösung optimal? Wenn $z^*_{\mathcal{N}} \geq 0$, dann stop, die aktuelle Lösung ist optimal. Dies gilt, da $x$ primal zulässig ist $x^*_{\mathcal{N}} = 0$ und $x^*_{\mathcal{B}} \geq 0$. Außerdem ist $z$ dual zulässig, d.h. $z^*_{\mathcal{B}} = 0$ und $z^*_{\mathcal{N}} \geq 0$. Insgesamt also

$$\left.\begin{array}{ll} x_i^* z_i^* = 0 & f.a.\, i \in \mathcal{N} \\ x_j^* z_j^* = 0 & f.a.\, j \in \mathcal{B} \end{array}\right\} \text{Komplementaritätsbedingungen.}$$

2. Schritt: Wähle Index $j \in \mathcal{N}$ mit $z_j^* < 0$ und $|z_j^*| \geq |z_i^*|$ für alle $i \in \mathcal{N}$ mit $z_i^* < 0$. Die Variable $x_j$ ist die Eintrittsvariable (damit wächst $f$ am meisten). Existiert mehr als einer, dann wähle den ersten möglichen, bzw. den kleinsten Index (Bland'sche Regel).

3. Schritt: Berechne die primale Schrittrichtung $\Delta x_{\mathcal{B}}$. Sei $e_k$ der $k$-te Einheitsvektor im $\mathbb{R}^n$; dann ist $\Delta x_{\mathcal{B}} = B^{-1}Ne_k$, wobei $k$ die $k$-te Position in der Menge $\mathcal{N}$ von $x_j$ bezeichnet, da aus (2.3.4) folgt:

$$x_{\mathcal{B}} = x^*_{\mathcal{B}} - B^{-1}Nte_k \quad \Rightarrow \quad \Delta x_{\mathcal{B}} = B^{-1}Ne_k.$$

4. Schritt: Berechne die primale Schrittlänge $t = \left(\max_{i\in\mathcal{B}} \frac{\Delta x_i}{x_i^*}\right)^{-1}$ mit Konvention $\frac{0}{0} = 0$; ist das Maximum kleiner oder gleich Null dann stop, das Primale ist unbeschränkt. Wir möchten $t \geq 0$ so groß wie möglich wählen, aber $x_{\mathcal{B}}$ darf nicht negativ werden, also $x^*_{\mathcal{B}} \geq t\Delta x_{\mathcal{B}}$. Wegen $i \in \mathcal{B}$, $x_i^* \geq 0$ und $t \geq 0$ gilt auch:

$$\frac{1}{t} \geq \frac{\Delta x_i}{x_i^*} \text{ f. a. } i \in \mathcal{B} \quad \Rightarrow \quad t = \left(\max_{i\in\mathcal{B}} \frac{\Delta x_i}{x_i^*}\right)^{-1}.$$

Ist das Maximum $\geq 0$, so können wir unser $f$ beliebig vergrößern (und damit ist das LP unbeschränkt). Gibt es mehrere $i \in \mathcal{B}$, für die das Maximum angenommen wird, so wähle den ersten möglichen Index. Jenes $i \in \mathcal{B}$, das gewählt wird, liefert die Austrittsvariable $x_i$.

5. Schritt: Berechne die duale Schrittrichtung $\Delta z_{\mathcal{N}} = -(B^{-1}N)^T e_l$, wobei $e_l$ der $l$-te Einheitsvektor im $\mathbb{R}^n$ ist und $l$ die $l$-te Position von $i \in \mathcal{B}$, erkennbar am dualen Dictionary.

6. Schritt: Berechne die duale Schrittlänge für die Austrittsvariable $z_j$:

$$s = \frac{z_j^*}{\Delta z_j}.$$

7. Schritt: Update aktuelle primale und duale Lösung:

$$\begin{array}{c} x_j^* = t, \quad \widetilde{x_{\mathcal{B}}} = x^*_{\mathcal{B}} - t\Delta x_{\mathcal{B}}, \quad x^*_{\mathcal{B}} = \widetilde{x_{\mathcal{B}}} + te_k \quad (e_k \text{ wie in 3}) \\ z_i^* = s, \quad \widetilde{z_{\mathcal{N}}} = z^*_{\mathcal{N}} - s\Delta z_{\mathcal{N}}, \quad z^*_{\mathcal{N}} = \widetilde{z_{\mathcal{N}}} + se_l \quad (e_l \text{ wie in 5}) \\ \mathcal{B} = \mathcal{B}\backslash\{i\} \cup \{j\}, \quad \mathcal{N} = \mathcal{N}\backslash\{j\} \cup \{i\}. \end{array}$$

### 2.3.3 Ein Beispiel

Gegeben sei:

$$\begin{aligned} \max \quad & 4x_1 + 3x_2 \\ \text{u.d.N.} \quad & x_1 - x_2 \leq 1 \\ & 2x_1 - x_2 \leq 3 \\ & x_2 \leq 5 \\ & x_1, x_2 \geq 0, \end{aligned}$$

dann ist:

$$A = \begin{pmatrix} 1 & -1 & 1 & 0 & 0 \\ 2 & -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{pmatrix}. \tag{2.3.5}$$

Die Initial Indice-Mengen sind gegeben durch

$$\mathcal{B} = \{3, 4, 5\} \quad \text{sowie} \quad \mathcal{N} = \{1, 2\}. \tag{2.3.6}$$

Die zugehörigen Teilmatrizen sind daher

$$B = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \text{und} \quad N = \begin{pmatrix} 1 & -1 \\ 2 & -1 \\ 0 & 1 \end{pmatrix}$$

und aus (4) und (5) sehen wir sofort, dass

$$x^*_{\mathcal{B}} = b = \begin{pmatrix} 1 \\ 3 \\ 5 \end{pmatrix}, \quad z^*_{\mathcal{N}} = -c_{\mathcal{N}} = \begin{pmatrix} -4 \\ -3 \end{pmatrix}$$

gilt.

I. Iteration:

1. Schritt: $x_{\mathcal{B}} \geq 0$, also primal zulässig; $z^*_{\mathcal{N}}$ hat negative Werte, d.h. die aktuelle Lösung ist nicht optimal.
2. Schritt: $z^*_1 = -4$ ist die größte negative, d.h. der Eintrittsindex ist $j = 1$.
3. Schritt: Da $j = 1$ auch an der ersten Position in $\mathcal{N}$ ist, gilt $k = 1$, also: $\Delta x_{\mathcal{B}} = B^{-1}Ne_k = Ne_k = \begin{pmatrix} 1 & -1 \\ 2 & -1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 0 \end{pmatrix}$.
4. Schritt: $t = \left(\max\{\frac{1}{1}, \frac{2}{3}, \frac{0}{5}\}\right)^{-1} = 1$; da das Maximum beim ersten Bruch angenommen wird, erhalten wir $i = 3$.
5. Schritt: Da $i = 3$ an der ersten Position in der Menge $\mathcal{B}$ ist, gilt $l = 1$, damit $\Delta z_{\mathcal{N}} = -(B^{-1}N)^T e_l = N^T e_l = -\begin{pmatrix} 1 & 2 & 0 \\ -1 & -1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} -1 \\ 1 \end{pmatrix}$.

6. Schritt: $s = \frac{z_j^*}{\Delta z_j} = \frac{-4}{-1} = 4$
7. Schritt:

$$x_1^* = 1, \qquad \widetilde{x_{\mathcal{B}}} = \begin{pmatrix} 1 \\ 3 \\ 5 \end{pmatrix} - 1 \begin{pmatrix} 1 \\ 2 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 5 \end{pmatrix}$$

$$z_3^* = 4, \qquad \widetilde{z_{\mathcal{N}}} = \begin{pmatrix} -4 \\ -3 \end{pmatrix} - 4 \begin{pmatrix} -1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ -7 \end{pmatrix}$$

$$\mathcal{B} = \{1, 4, 5\}, \qquad \mathcal{N} = \{3, 2\}$$

$$B = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \qquad N = \begin{pmatrix} 1 & -1 \\ 0 & -1 \\ 0 & 1 \end{pmatrix},$$

$$x_{\mathcal{B}}^* = \begin{pmatrix} x_1^* \\ x_4^* \\ x_5^* \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 5 \end{pmatrix}, \qquad z_{\mathcal{N}}^* = \begin{pmatrix} z_3^* \\ z_2^* \end{pmatrix} = \begin{pmatrix} 4 \\ -7 \end{pmatrix}$$

II. Iteration

1. Schritt: $z_{\mathcal{N}}^*$ hat negative Werte, d.h. die aktuelle Lösung ist nicht optimal.
2. Schritt: Da $z_2^* = -7$ ist, erhalten wir den Eintrittsindex $j = 2$.
3. Schritt: $\Delta x_{\mathcal{B}} = B^{-1} N e_k = N e_k =$

$$\begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}^{-1} \begin{pmatrix} 1 & -1 \\ 0 & -1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -1 \\ 1 \\ 1 \end{pmatrix}$$

4. Schritt: $t = \left(\max\{\frac{-1}{1}, \frac{1}{1}, \frac{1}{5}\}\right)^{-1} = 1$; zweiter Bruch, also $i = 4$.
5. Schritt: $\Delta z_{\mathcal{N}} = -(B^{-1}N)^T e_l = N^T e_l =$

$$-\begin{pmatrix} 1 & 0 & 0 \\ -1 & -1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}^{-1} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 2 \\ -1 \end{pmatrix}$$

6. Schritt: $s = \frac{z_j^*}{\Delta z_j} = \frac{-7}{-1} = 7$

7. Schritt:

$$x_2^* = 1, \qquad \widetilde{x_{\mathcal{B}}} = \begin{pmatrix} 1 \\ 1 \\ 5 \end{pmatrix} - 1 \begin{pmatrix} -1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \\ 4 \end{pmatrix}$$

$$z_4^* = 7, \qquad \widetilde{z_{\mathcal{N}}} = \begin{pmatrix} 4 \\ -7 \end{pmatrix} - 7 \begin{pmatrix} 2 \\ -1 \end{pmatrix} = \begin{pmatrix} -10 \\ 0 \end{pmatrix}$$

$$\mathcal{B} = \{1, 2, 5\}, \qquad \mathcal{N} = \{3, 4\}$$

$$B = \begin{pmatrix} 1 & -1 & 0 \\ 2 & -1 & 0 \\ 0 & 1 & 1 \end{pmatrix}, \qquad N = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix},$$

$$x_{\mathcal{B}}^* = \begin{pmatrix} x_1^* \\ x_2^* \\ x_5^* \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \\ 4 \end{pmatrix}, \qquad z_{\mathcal{N}}^* = \begin{pmatrix} z_3^* \\ z_4^* \end{pmatrix} = \begin{pmatrix} -10 \\ 7 \end{pmatrix}$$

III. Iteration

1. Schritt: $z_{\mathcal{N}}^*$ hat negative Werte, d.h. die aktuelle Lösung ist nicht optimal.
2. Schritt: Da $z_3^* = -10$ erhalten wir den Eintrittsindex $j = 3$.
3. Schritt: $\Delta x_{\mathcal{B}} = B^{-1} N e_k = N e_k =$

$$\begin{pmatrix} 1 & -1 & 0 \\ 2 & -1 & 0 \\ 0 & 1 & 1 \end{pmatrix}^{-1} \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} -1 \\ -2 \\ 2 \end{pmatrix}$$

4. Schritt: $t = \left( \max\{\frac{-1}{2}, \frac{-2}{1}, \frac{2}{4}\} \right)^{-1} = 2$; dritter Bruch, also $i = 5$
5. Schritt: $\Delta z_{\mathcal{N}} = -(B^{-1}N)^T e_l = N^T e_l =$

$$-\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 2 & 0 \\ -1 & -1 & 1 \\ 0 & 0 & 1 \end{pmatrix}^{-1} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -2 \\ 1 \end{pmatrix}$$

6. Schritt: $s = \frac{z_j^*}{\Delta z_j} = \frac{-10}{-2} = 5$

7. Schritt:

$$x_3^* = 2, \qquad \widetilde{x_{\mathcal{B}}} = \begin{pmatrix} 2 \\ 1 \\ 4 \end{pmatrix} - 2 \begin{pmatrix} -1 \\ -2 \\ 2 \end{pmatrix} = \begin{pmatrix} 4 \\ 5 \\ 0 \end{pmatrix}$$

$$z_5^* = 5, \qquad \widetilde{z_{\mathcal{N}}} = \begin{pmatrix} -10 \\ 7 \end{pmatrix} - 5 \begin{pmatrix} -2 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 2 \end{pmatrix}$$

$$\mathcal{B} = \{1, 2, 3\}, \qquad \mathcal{N} = \{5, 4\}$$

$$B = \begin{pmatrix} 1 & -1 & 1 \\ 2 & -1 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \qquad N = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix},$$

$$x_{\mathcal{B}}^* = \begin{pmatrix} x_1^* \\ x_2^* \\ x_3^* \end{pmatrix} = \begin{pmatrix} 4 \\ 5 \\ 2 \end{pmatrix}, \qquad z_{\mathcal{N}}^* = \begin{pmatrix} z_5^* \\ z_4^* \end{pmatrix} = \begin{pmatrix} 5 \\ 2 \end{pmatrix}$$

IV. Iteration 1. Schritt: alle $z_{\mathcal{N}}^*$ sind nicht negativ, d.h. die aktuelle Lösung ist optimal

$$f^* = c_{\mathcal{B}}^T x_{\mathcal{B}}^* = (c_1, c_2, c_3) \begin{pmatrix} x_1^* \\ x_2^* \\ x_3^* \end{pmatrix} = 4x_1^* + 3x_2^* = 4 * 4 + 3 * 5 = 31.$$

### 2.3.4 Dualer Simplex-Algorithmus

Fast analog zum primalen können wir auch den dualen Simplex-Algorithmus formulieren. Die Grundidee des dualen Simplex-Algorithmus ist es, den primalen Simplex-Algorithmus auf das duale LP anzuwenden, ohne dieses explizit zu dualisieren. Hier der direkte Vergleich der beiden Algorithmen:

| Primaler Simplex | Dualer Simplex |
|---|---|
| Angenommen $x^*_{\mathcal{B}} \geq 0$ | Angenommen $z^*_{\mathcal{N}} \geq 0$ |
| while $(z^*_{\mathcal{N}} \ngeq 0)$ do | while $(x^*_{\mathcal{B}} \ngeq 0)$ do |
| $j \in \{j \in \mathcal{N} : z^*_j < 0\}$ | $i \in \{i \in \mathcal{B} : x^*_i < 0\}$ |
| $\Delta x_{\mathcal{B}} = B^{-1}Ne_k$ | $\Delta z_{\mathcal{N}} = -(B^{-1}N)^T e_l$ |
| $t = \left(\max_{i\in\mathcal{B}} \frac{\Delta x_i}{x^*_i}\right)^{-1}$ | $s = \left(\max_{j\in\mathcal{N}} \frac{\Delta z_j}{z^*_j}\right)^{-1}$ |
| $i \in \operatorname{argmax}_{i\in\mathcal{B}} \frac{\Delta x_i}{x^*_i}$ | $j \in \operatorname{argmax}_{j\in\mathcal{N}} \frac{\Delta z_j}{z^*_j}$ |
| $\Delta z_{\mathcal{N}} = -(B^{-1}N)^T e_l$ | $\Delta x_{\mathcal{B}} = B^{-1}Ne_k$ |
| $s = \frac{z^*_j}{\Delta z_j}$ | $t = \frac{x^*_i}{\Delta x_i}$ |
| $x^*_j = t$ | $x^*_j = t$ |
| $\widetilde{x_{\mathcal{B}}} = x^*_{\mathcal{B}} - t\Delta x_{\mathcal{B}}$ | $\widetilde{x_{\mathcal{B}}} = x^*_{\mathcal{B}} - t\Delta x_{\mathcal{B}}$ |
| $x^*_{\mathcal{B}} = \widetilde{x_{\mathcal{B}}} + te_k$ | $x^*_{\mathcal{B}} = \widetilde{x_{\mathcal{B}}} + te_k$ |
| $z^*_i = s$ | $z^*_i = s$ |
| $\widetilde{z_{\mathcal{N}}} = z^*_{\mathcal{N}} - s\Delta z_{\mathcal{N}}$ | $\widetilde{z_{\mathcal{N}}} = z^*_{\mathcal{N}} - s\Delta z_{\mathcal{N}}$ |
| $z^*_{\mathcal{N}} = \widetilde{z_{\mathcal{N}}} + se_l$ | $z^*_{\mathcal{N}} = \widetilde{z_{\mathcal{N}}} + se_l$ |
| $\mathcal{B} = \mathcal{B}\backslash\{i\} \cup \{j\}$ | $\mathcal{B} = \mathcal{B}\backslash\{i\} \cup \{j\}$ |
| $\mathcal{N} = \mathcal{N}\backslash\{j\} \cup \{i\}$ | $\mathcal{N} = \mathcal{N}\backslash\{j\} \cup \{i\}$ |
| end while | end while |

### 2.3.5 Zwei-Phasen Algorithmus

Gegeben sei ein LP in Normalform II wie in (2.3.2) mit $\mathcal{B} = \{n+1, \ldots, n+m\}$, und $\mathcal{N} = \{1, \ldots, n\}$. Da $A = [N\, B]$ ist

$$N = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}, \quad B = \begin{pmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \\ 0 & \cdots & 1 \end{pmatrix}, \quad c_{\mathcal{N}} = \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix}, \quad c_{\mathcal{B}} = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}$$

$$x^*_{\mathcal{B}} = B^{-1}b = b, \quad z^*_{\mathcal{N}} = (B^{-1}N)^T c_{\mathcal{B}} - c_{\mathcal{N}} = -c_{\mathcal{N}}, \quad f^* = 0.$$

Also ist das initial-Dictionary gegeben durch

$$\begin{aligned} f &= c^T_{\mathcal{N}} x_{\mathcal{N}} \\ x_{\mathcal{B}} &= b - N x_{\mathcal{N}}. \end{aligned}$$

Wir betrachten vier Fälle:

1. Fall: Es gilt $x_{\mathcal{B}}$ nicht negativ, $c_{\mathcal{N}}$ nicht positiv, damit ist die Lösung des LP optimal.
2. Fall: Es gilt $x_{\mathcal{B}}$ nicht negativ, aber $c_{\mathcal{N}}$ teilweise positiv, also ist die Lösung des LP primal zulässig; starte primalen Simplex-Algorithmus um das LP zu lösen.
3. Fall: Es gilt $x_{\mathcal{B}}$ negativ aber $c_{\mathcal{N}}$ nicht positiv, die Lösung des LP ist dual zulässig; starte dualen Simplex-Algorithmus um das LP zu lösen.

4. Fall: Manche $x_{\mathcal{B}}$ sind negativ und manche $c_{\mathcal{N}}$ sind positiv, dann ist das LP weder primal, noch dual zulässig.
   Wir verwenden eine 2 Phasen-Methode:
   Ersetze $c_{\mathcal{N}}$ durch einen nicht-positiven Vektor, damit ist das modifizierte Problem dual zulässig; der duale Simplex findet eine optimale Lösung, welche in die ursprüngliche Zielfunktion eingesetzt wird. Danach wird das primal Zulässige mit dem primalem Simplex-Algorithmus gelöst. Analog können wir natürlich auch $x_{\mathcal{B}}$ durch einen positiven Vektor ersetzen und das dann primal zulässige Problem mit dem primalen Simplex lösen. Diese optimale Lösung wird in die ursprüngliche Zielfunktion eingesetzt und das dann dual zulässige Problem mit dem dualen Simplex gelöst.

Wir betrachten für den Punkt 4 folgendes Beispiel:

**Beispiel 2.3.1** Gegeben sei das Problem

$$\begin{aligned} &\max c^T x \\ &\text{u.d.N. } Px \leq b \end{aligned}$$

mit

$$c = \begin{pmatrix} 3 \\ 1 \\ -5 \end{pmatrix}, \quad P = \begin{pmatrix} 1 & 0 & -4 \\ 1 & 3 & -1 \end{pmatrix} \quad \text{und} \quad b = \begin{pmatrix} -1 \\ 11 \end{pmatrix}.$$

Dann ist das initial-Dictionary gegeben durch:

$$A = \begin{pmatrix} 1 & 0 & -4 & 1 & 0 \\ 1 & 3 & -1 & 0 & 1 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad N = \begin{pmatrix} 1 & 0 & -4 \\ 1 & 3 & -1 \end{pmatrix},$$

$$\mathcal{B} = \{4, 5\}, \quad \mathcal{N} = \{1, 2, 3\},$$

$$x_{\mathcal{B}} = b = \begin{pmatrix} 0 \\ 11 \end{pmatrix} \quad \text{und} \quad z_{\mathcal{N}} = -c_{\mathcal{N}} = \begin{pmatrix} -3 \\ -1 \\ 5 \end{pmatrix}.$$

Da manche $x_{\mathcal{B}}$ negativ und manche $c_{\mathcal{N}}$ positiv sind, sehen wir sofort, dass dieses Problem weder primal noch dual zulässig ist und wir somit im vierten Fall sind.

**Phase I**: Wir substituieren

$$\widetilde{x_{\mathcal{B}}} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

und lösen das modifizierte und primal zulässige Problem und erhalten das Dictionary:

$$A = \begin{pmatrix} 1 & 0 & -4 & 1 & 0 \\ 1 & 3 & -1 & 0 & 1 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & -4 \\ 1 & -1 \end{pmatrix}, \quad N = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 3 & 1 \end{pmatrix},$$

$$\mathcal{B} = \{1, 3\}, \quad \mathcal{N} = \{4, 2, 5\},$$

$$\widetilde{x_{\mathcal{B}}} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad \text{und} \quad z_{\mathcal{N}} = -c_{\mathcal{N}} = \begin{pmatrix} 2/3 \\ 6 \\ 7/3 \end{pmatrix}$$

**Phase II**: Wir müssen $\widetilde{x_{\mathcal{B}}}$ zurück substituieren. Wir wissen, dass stets $x^*_{\mathcal{B}} = B^{-1}b$ gilt (vgl. 2.3.2 (1)). Also ist

$$x_{\mathcal{B}} = B^{-1} = \begin{pmatrix} 15 \\ 4 \end{pmatrix}$$

und damit ist unser dual zulässiges Dictionary gegeben durch:

$$A = \begin{pmatrix} 1 & 0 & -4 & 1 & 0 \\ 1 & 3 & -1 & 0 & 1 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & -4 \\ 1 & -1 \end{pmatrix}, \quad N = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 3 & 1 \end{pmatrix},$$

$$\mathcal{B} = \{1, 3\}, \quad \mathcal{N} = \{4, 2, 5\},$$

$$\widetilde{x_{\mathcal{B}}} = \begin{pmatrix} 15 \\ 4 \end{pmatrix} \quad \text{und} \quad z_{\mathcal{N}} = -c_{\mathcal{N}} = \begin{pmatrix} 2/3 \\ 6 \\ 7/3 \end{pmatrix}.$$

Die optimale Lösung hiervon lautet:

$$x^*_1 = 15,\ x^*_2 = 0,\ x^*_3 = 4 \quad \text{und} \quad f = 25.$$

### 2.3.6 Negativ-transponierte Eigenschaft

Betrachte das LP mit seinem Dualen

$$\begin{array}{ll} \max c^T x & \min b^T y \\ \text{u.d.N. } Ax \le b \quad \text{(P)} & \text{u.d.N. } A^T y \ge c \quad \text{(D)} \\ x \ge 0 & y \ge 0. \end{array}$$

Sei $w$ der Vektor, der die Schlupfvariablen von (P) repräsentiert, und $z$ der von (D), damit erhalten wir:

$$\begin{array}{ll} \max c^T x & \min b^T y \\ \text{u.d.N. } Ax + w = b \quad \text{(P)} & \text{u.d.N. } A^T y - z = c \quad \text{(D)} \\ x, w \ge 0 & y, z \ge 0 \end{array}$$

Wir schreiben $\bar{A} = [A\, I]$, $\bar{c} = \begin{bmatrix} c \\ 0 \end{bmatrix}$, $\bar{x} = \begin{bmatrix} x \\ w \end{bmatrix}$, $\hat{A} = [-I\, A^T]$, $\hat{b} = \begin{bmatrix} 0 \\ b \end{bmatrix}$, $\hat{y} = \begin{bmatrix} z \\ y \end{bmatrix}$ dann gilt:

$$\begin{array}{ccc} \max \bar{c}^T \bar{x} & & \min \hat{b}^T \hat{y} \\ \text{u.d.N. } \bar{A}\bar{x} = b \quad \text{(P)} & & \text{u.d.N. } \hat{A}^T \hat{y} = c \quad \text{(D)} \\ \bar{x} \geq 0 & & \hat{y} \geq 0. \end{array}$$

Beachte $\bar{A} = [A\, I] \in \mathbb{R}^{m \times (n+m)}$ und $\hat{A} = [-I\, A^T] \in \mathbb{R}^{n \times (n+m)}$. Nach einigen Schritten im Simplex-Algorithmus können wir aber festhalten, dass bis auf Umsortieren der Spalten gilt:

$$\bar{A} = [A\, I] = [\bar{N}\, \bar{B}] \qquad \text{und} \qquad \hat{A} = [-I\, A^T] = [\hat{B}\, \hat{N}].$$

Der primale Simplex-Algorithmus produziert die Matrix $\bar{B^{-1}}\bar{N}$, während der duale die Matrix $\hat{B^{-1}}\hat{N}$ produziert.

**Satz 2.3.2** Die beiden Matrizen $\bar{A}$ und $\hat{A}$ sind die jeweiligen negativ transponierten zueinander. [2]

*Beweis:* Es gelten:

$$\bar{A}\hat{A}^T = [A\, I][-I\, A]^T = [\bar{N}\, \bar{B}] \begin{bmatrix} \hat{B}^T \\ \hat{N}^T \end{bmatrix} = \bar{N}\hat{B}^T + \bar{B}\hat{N}^T \quad \text{und}$$
$$\bar{A}\hat{A}^T = [A\, I][-I\, A]^T = -A + A = 0$$

$$\begin{aligned} \Rightarrow \quad \bar{N}\hat{B}^T + \bar{B}\hat{N}^T &= 0 \\ \bar{N}\hat{B}^T &= -\bar{B}\hat{N}^T \\ \bar{B}^{-1}\bar{N} &= -\hat{N}^T \hat{B^{-1}}^T \\ \bar{B}^{-1}\bar{N} &= -(\hat{B^{-1}}\hat{N})^T. \end{aligned}$$

□

Damit sehen wir, dass es überhaupt nicht notwendig ist, das duale vom primalen Problem aufzuschreiben. Wir können beim Simplex-Algorithmus jederzeit das duale Problem betrachten.

## 2.4 Nachbemerkungen

In diesem Kapitel wollen wir noch ein paar Nachbemerkungen machen, unter anderem zu Problemen bei der Implementierung und zur Sensitivitätsanalyse.

[2] Der Beweis ist in [Van14], Kapitel 6.6, zu finden.

### 2.4.1 Entartete Basispunkte und Pivotstrategien

**Definition 2.4.1** Eine zulässige (Basis-)Lösung eines gegebenen LPs heißt entartet oder degeneriert, falls mindestens eine der Basisvariablen den Wert 0 hat (gilt also $x_{\mathcal{B}} > 0$ so ist die Basis nicht entartet).

Ist ein gegebenes LP degeneriert, kann es dazu kommen, dass der Simplex-Algorithmus anfängt in einer Ecke zu kreisen (womit nicht mehr garantiert ist, dass der Algorithmus nach endlich vielen Schritten terminiert). Dies kann mit geeigneten Pivotstrategien behoben werden.

Wir betrachten nur den primalen Simplex-Algorithmus, da sich die Pivotstrategien auch auf den dualen Simplex-Algorithmus übertragen lassen.
Die Bland'sche Pivotregel, die wir beim Formulieren des Algorithmus verwendet haben, lautet: Wir nehmen an, dass wir im 2. Schritt einer Iteration angekommen sind und die Eintrittsvariable wählen sollen. Wir wählen $z_j^* < 0$ mit $j \in \mathcal{N}$, wobei wir das kleinste wählen. Gibt es nun mehrere solche $z_j^*$, so wählen wir jenes mit dem kleinsten Index. Analog im 4. Schritt wählen wir die Austrittsvariable $i \in \mathcal{B}$ für die das Maximum bei der Berechnung der primalen Schrittweite erreicht wurde. Gibt es auch hier mehrere Möglichkeiten, so wählen wir wieder jene Variable mit dem kleinstem Index. Man kann zeigen, dass der Simplex-Algorithmus dadurch immer terminiert und somit das Kreisen in einer Ecke verhindert wird.

Entscheidend für die Geschwindigkeit bzw. die Anzahl der Iterationen des Simplex-Algorithmus ist die Pivotstrategie, von der es unzählige gibt. Hier eine kurze Auswahl alternativer Pivotstrategien:[3]

1. Das steepest-edge pricing: Diese Pivotstrategie kombiniert die Wahl der Ein- und Austrittsvariable, sodass der Wert der Zielfunktion möglichst große Zuwächse in jeder Iteration generiert. Dies führt zu einem größeren Rechenaufwand in jeder Iteration, aber führt mit weniger Iterationen zum Ergebnis.

2. Das devex pricing von Paula Harris aus dem Jahr 1973 [Har73]: Dies ist eine Approximation von steepest edge, wobei diese vor der Auswahl der Ein- und Austrittsvariable auf eine einheitliche Norm skaliert werden, um eine größere Aussagekraft über die getroffene Wahl zu erhalten.

3. Das partial pricing: Hier werden die Variablen in Blöcke unterteilt. Die Idee ist nun, den Simplex-Algorithmus auf jeden dieser Blöcke einzeln anzuwenden, bis kein weiteres Wachstum der Zielfunktion mehr erreicht werden kann. Beim Lösen des jeweiligen Blockes wird einer der beiden oben genannten Verfahren verwendet.

4. Die lexikographische Zeilenauswahl: Sie garantiert, im Gegensatz zu den anderen eben genannten, dass der Simplex-Algorithmus nicht ins Kreisen gerät und damit eine Terminierung. Die Idee ist ähnlich wie bei der Bland'schen Regel. Es wird unter allen möglichen Kandidaten bei der Wahl der Eintritts- bzw. Austrittsvariable die eindeutig lexikographisch kleinste gewählt. Analog wie die Bland'sche Pivotregel ist die lexikographische Zeilenauswahl vergleichsweise langsam.

[3] Eine ausführliche und detaillierte Darstellung verschiedener Pivotstrategien ist in [Wun96], Abschnitt 1.6 zu finden. Dort werden alle hier genannten, mit Ausnahme der lexikographischen Zeilenauswahl, ausführlich erläutert. Die lexikographischen Zeilenauswahl ist in [Van14], Kapitel 3.3 zu finden.

### 2.4.2 Geschwindigkeit und Terminierung des Simplex-Algorithmus

Jede Pivotstrategie hat Vor- und Nachteile. Die Bland'sche Regel etwa garantiert zwar, dass der Algorithmus nicht ins Kreisen gerät, ist aber in der Regel sehr langsam. Lang wurde versucht, eine allgemeine Pivot-Strategie zu finden, welche den Simplex-Algorithmus auch besonders effizient macht, bzw. garantiert, dass er in polynomieller Laufzeit terminiert. Es gibt einige Beispiele, die zeigen, dass die Geschwindigkeit exponentiell steigt. [4] Die endliche Anzahl an Ecken garantiert jedoch immer ein Terminieren des Algorithmus (sofern keine entarteten Punkte vorhanden sind oder eine entsprechende Regel zum Verarbeiten dieser benutzt wird). Das Beispiel von Klee und Minty ist das Folgende:

$$\begin{aligned} & \max \sum_{j=1}^{n} 10^{n-j} x_j & & \\ \text{u.d.N.} \quad & 2\sum_{j=1}^{i-1} 10^{i-j} x_j + x_i \leq 100^{i-1} & & i = 1, \ldots, n \\ & x_j \geq 0 & & j = 1, \ldots, n. \end{aligned}$$

Die ersten drei Ungleichungen sind:

$$\begin{aligned} x_1 &\leq 1 \\ 20x_1 + x_2 &\leq 100 \\ 200x_1 + 20x_2 + x_3 &\leq 10000. \end{aligned}$$

Klee und Minty haben gezeigt, dass der Simplex-Algorithmus in diesem Fall $2^n - 1$ Iterationen braucht. Dennoch ist es in der Praxis so, dass dieses "worst-case-Szenario" eher selten vorkommt. [5]

### 2.4.3 Sensitivitätsanalyse

Die Frage, die wir uns hier stellen wollen, ist, was passiert, wenn wir in einem linearen Optimierungsproblem die Daten verändern, was in der Praxis durchaus häufig vorkommt. Betrachten wir ein LP in folgender Form und nehmen an, dass wir bereits eine optimale Lösung $\bar{x}$ gefunden haben:

$$\begin{aligned} & \max c^T x \\ \text{u.d.N.} \quad & Ax \leq b \\ & x \geq 0. \end{aligned}$$

Wir formulieren folgende Eigenschaften über das „warm-starting“ des Simplex-Algorithmus. [6]

Was passiert bei

[4] Das Erste stammt von Victor Klee und George Minty - der sogenannte Klee und Minty Würfel [KlMi72].

[5] Unter bestimmten Annahmen an das LP konnte in den 80ern von Borgwardt und anderen gezeigt werden, dass der Simplex-Algorithmus eine polynomielle Laufzeit hat und solche Beispiele wie der Klee und Minty Würfel in der Praxis sehr selten vorkommen. Vergleiche dazu [Bor04] und [Bor14].

[6] Alle hier getroffenen Aussagen werden in [Due08], Abschnitt 5.6 bewiesen.

1. Änderung der Zielfunktion $c$:
   Es können zwei Fälle auftreten; in beiden bleibt die Basis primal zulässig. Im ersten Fall bleibt sie auch dual und die Basis $\mathcal{B}$ liefert weiterhin unsere optimale Lösung (wobei sich eventuell der Wert der Zielfunktion ändert). Im zweiten Fall ist sie nicht mehr dual zulässig, wir können aber mit dem primalen Simplex-Algorithmus eine neue Lösung finden.

2. Änderung der rechten Seite $b$:
   Man kann zeigen, dass die optimale Lösung eines LP stetig von der Änderung der rechten Seite abhängt.

3. Änderung eines Eintrags in der Matrix $A$:
   Es gibt zwei Fälle. Entweder bleibt die Lösung optimal oder sie ist nicht mehr dual zulässig, kann dann aber wieder mit dem primalen Simplex-Algorithmus gelöst werden.

4. Hinzufügen einer neuen Variable:
   Es kommt eine neue zu suchende Größe $x_i$ dazu, sowie ein neuer Zielfunktionskoeffizient $c_i$ und eine Spalte in der Matrix $A$. Es können wieder zwei Fälle auftreten. Entweder die Lösung bleibt dual zulässig und damit optimal, oder es kann mit dem primalen Simplex-Algorithmus eine Neue gefunden werden.

5. Hinzufügen einer neuen Nebenbedingung:
   Wir möchten nun eine neue Nebenbedingung hinzufügen, etwa

   $$A_{m+1}x_{m+1} \leq b_{m+1}.$$

   Dadurch erhalten wir

   $$\begin{array}{lcl} \max & c^T x & \\ \text{u.d.N.} & \begin{bmatrix} A \\ A_{m+1} \end{bmatrix} x & \leq \begin{bmatrix} b \\ b_{m+1} \end{bmatrix} \\ & x & \geq 0. \end{array}$$

   Entweder unser $\bar{x}$ erfüllt auch die neue Nebenbedingung, dann ist nichts zu tun. Andernfalls setze:

   - $\mathcal{B} = \bar{\mathcal{B}} \cup \{n+m+1\}$
   - $x_{\mathcal{B}} = \begin{bmatrix} \bar{x_{\mathcal{B}}} \\ -(N_{m+1}\bar{x}) + b_{m+1} \end{bmatrix}$
   - $A = \begin{bmatrix} A & 0 \\ N_{m+1} & 1 \end{bmatrix}$
   - $B = (A_{\mathcal{B}_1}, \cdots, A_{\mathcal{B}_{n+1}})$, $N = (A_{\mathcal{N}_1}, \cdots, A_{\mathcal{N}_m})$
   - Und der Rest bleibt wie gehabt; löse das dual zulässige Problem mit dem dualen Simplex.

Da wir später im Kapitel über das Branch-and-Bound-Verfahren regelmäßig neue Nebenbedingungen hinzufügen, betrachten wir zu Punkt 5 ein Beispiel.

**Beispiel 2.4.2** Gegeben seien:

$$\begin{aligned} \max \quad & 17x_1 + 12x_2 & \\ \text{u.d.N.} \quad & 10x_1 + 7x_2 & \leq 40 \\ & x_1 + x_2 & \leq 5 \\ & x_1, x_2 & \geq 0. \end{aligned}$$

Simplex liefert:

$$A = \begin{pmatrix} 10 & 7 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{pmatrix}, \quad \bar{B} = \begin{pmatrix} 10 & 7 \\ 1 & 1 \end{pmatrix}, \quad \bar{N} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix},$$

$$\bar{x} = \begin{pmatrix} 1.66 \\ 3.33 \end{pmatrix}, \quad x_{\bar{\mathcal{B}}} = \begin{pmatrix} 1.66 \\ 3.33 \end{pmatrix}, \quad z_{\bar{\mathcal{N}}} = \begin{pmatrix} 1.66 \\ 0.33 \end{pmatrix},$$

$$\text{sowie} \quad \bar{\mathcal{B}} = \{1, 2\}, \quad \bar{\mathcal{N}} = \{3, 4\}.$$

Wir wollen die neue Nebenbedingung $x_1 \leq 1$ einfügen. Setze:

- $\mathcal{B} = \bar{\mathcal{B}} \cup \{n + m + 1\} = \{1, 2\} \cup \{5\} = \{1, 2, 5\}$
- $x_{\mathcal{B}} = \begin{bmatrix} x_{\bar{\mathcal{B}}} \\ -(N_{m+1}\bar{x}) + b_{m+1} \end{bmatrix} = \begin{pmatrix} 1.66 \\ 3.33 \\ -0.66 \end{pmatrix}$
- $A = \begin{bmatrix} A & 0 \\ N_{m+1} & 1 \end{bmatrix} = \begin{pmatrix} 10 & 7 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix}$
- $B = \begin{pmatrix} 10 & 7 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}, N = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$
- Den Rest lassen wir wie gehabt und lösen dann mit dem dualem Simplex.

Wir erhalten nach bereits einer Iteration:

$$x_{\text{opt}} = \begin{pmatrix} 1 \\ 4 \end{pmatrix} \quad \text{und} \quad f = 65.$$

### 2.4.4 Inverse bestimmen

Ein großes Problem im Algorithmus ist das Bestimmen der Inverse von $B$. Die zwei bekanntesten Verfahren um dieses Problem zu lösen, sind die LR-Zerlegung und die Produktdarstellung. [7] Es gibt inzwischen verschiedene Methoden, die etwa in der ersten Iteration eine LR-Zerlegung durchführen und diese im weiteren Verlauf nutzen, um die Inverse von $B$ zu berechnen. Einige weniger effiziente aber genauere Verfahren berechnen in jedem Schritt eine neue LR-Zerlegung.

[7] Beide Verfahren werden ausführlich im achten Kapitel von [Van14] beschrieben.

Bei der Produktdarstellung wird $B^{-1}$ in ein Produkt zerlegt, dass in jeder Iteration durch eine Matrix ergänzt wird und die einzelnen Pivotschritte durch Matrizen beschreibt. Man kann zeigen, dass dann gilt:

$$B^{-1} = T_0 \cdots T_k,$$

wobei die $T_i$ leicht zu bestimmen sind bzw. am Pivotschritt abgelesen werden können. Es gilt:

$$T_i = I - \frac{(\Delta x_{\mathcal{B}} - e_k)e_k^T}{\Delta x_k} \qquad i \in \{1, \dots k\},$$

wobei $\Delta x_{\mathcal{B}}$, $e_k$ und $\Delta x_k$ die jeweiligen Größen der i-ten Iteration sind.

# 3 Branch-and-Bound

Das Branch-and-Bound-Verfahren ist eine Methode um ein gegebenes Optimierungsproblem, bei dem eine ganzzahlige Lösung gesucht wird, zu lösen. Da Branch-and-Bound selbst kein eigenes Verfahren ist, sondern nur eine mögliche Methode mit dem Problem umzugehen, gibt es für die entsprechenden Probleme passende Algorithmen. Wir befassen uns hier mit linearen ganzzahligen Optimierungsproblemen. Wir orientieren uns dabei hauptsächlich an Kapitel 3 in [Sie96] und Kapitel 6.4 in [Dom15]. Das Verfahren selbst wurde erstmals 1960 von A. H. Land und A. G. Doig vorgestellt, während R. J. Dakin 1965 einen Algorithmus angab, der einfach zu implementieren war. [8]

## 3.1 Einführendes Beispiel

Das folgende Beispiel stammt aus Kapitel 3 in [Sie96]. Die Firma „KäMi“ produziert neben Milch hauptsächlich Käse. Ein Teil des Transportes erledigt die Firma selbst, der Rest wurde verlagert. Der aktuelle Fuhrpark von „KäMi“ ist veraltet und muss erneuert werden. Zwei Fahrzeugtypen stehen in der engeren Wahl. Mit einem Wagen vom Typ $A$ kann ausschließlich 100 ($\times$ 100 kg) Käse transportiert werden, während mit einem Wagen vom Typ $B$ sowohl Milch als auch Käse mit maximal 50 ($\times$ 100 kg) Käse und 20 ($\times$ 100 l) Milch transportiert werden kann. Der Kauf eines Fahrzeuges vom Typ $A$ ergibt ein Ersparnis von 1000 ($\times$ 1 €) pro Monat gegenüber dem Transport über eine externe Firma, beim Typ $B$ sind es 700 ($\times$ 1 €) pro Monat. Selbstverständlich möchte die „Kämi“ ihr Einsparpotential maximieren. Damit erhalten wir, wenn wir mit $x_1$ die Anzahl der Fahrzeuge vom Typ $A$ und mit $x_2$ die Anzahl der Fahrzeuge vom Typ $B$ bezeichnen, das Zielfunktional:

$$\max 1000x_1 + 700x_2.$$

Da die Nachfrage nach Käse und Milch regelmäßigen Schwankungen unterliegt, hat das Management von „KäMi“ entschieden, dass die Gesamtkapazität von Fahrzeugen, die beschafft werden sollen, nicht die minimale Nachfrage pro Tag übersteigen soll. Im Falle von Käse sind das 2425 ($\times$ 100 kg) pro Tag und im Falle von Milch sind das 510 ($\times$ 100 kg) pro Tag. Damit ergeben sich folgende Nebenbedingungen:

$$\begin{aligned} 100x_1 + 50x_2 &\leq 2425 \\ 20x_2 &\leq 510. \end{aligned}$$

Weiterhin ist zu beachten, dass die Variablen $x_1$ und $x_2$ ganzzahlig sein müssen, da es keinen Sinn ergibt nur Teile eines Fahrzeuges zu kaufen. Damit erhalten wir das ganzzahlige Optimierungsproblem „KäMi“:

[8] Vergleiche hierzu [LaDo60] und [Dak65]

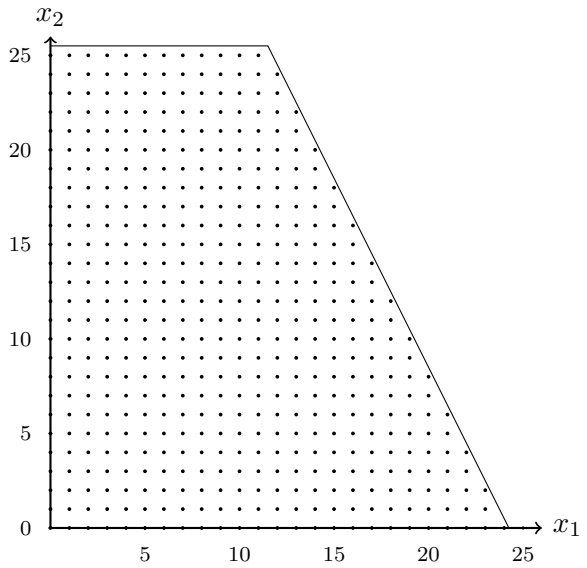



Abbildung 3.1: Der zulässige Bereich von dem Modell „KäMi“

$$\begin{aligned} \max \quad & 1000x_1 + 700x_2 \\ \text{u.d.N.} \quad & 100x_1 + 50x_2 \leq 2425 \\ & 20x_2 \leq 510 \\ & x_1, x_2 \geq 0, \text{ganzzahlig.} \end{aligned}$$

In Abbildung 3.1 ist der zulässige Bereich dargestellt. Dabei ist zu beachten, dass sich der zulässige Bereich nur über die ganzzahligen Punkte erstreckt. Das vorgestellte Simplex-Verfahren findet nun zwar eine Lösung (die in einer Ecke liegt), diese muss aber nicht mehr ganzzahlig sein, wie wir auf dem Bild sehen können. Einfaches Abrunden wird auch nicht genügen. Hier liegt das Optimum (wenn wir die Ganzzahligkeitsbedingung ignorieren) in der Ecke $(x_1, x_2) = (11\frac{1}{2}, 25\frac{1}{2})$ mit Maximum $f = 29350$. Würden wir eine der Komponenten aufrunden, verließen wir den zulässigen Bereich, Abrunden liefert den Punkt $(x_1, x_2) = (11, 25)$ mit Wert der Zielfunktion $f = 28500$. Aber es ist leicht zu sehen, dass etwa der Punkt $(x_1, x_2) = (12, 24)$ einen höheren Wert der Zielfunktion erreicht ($f = 28800$) und ebenfalls zulässig ist. Wir halten zunächst folgende Bemerkung fest:

**Bemerkung 3.1.1** Für beliebige $x$, $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$ und $A \in \mathbb{R}^{m \times n}$ gilt:

$$\max\{c^T x \,|\, Ax \leq b,\, x \geq 0,\, x \text{ ganzzahlig}\} \leq \max\{c^T x \,|\, Ax \leq b,\, x \geq 0,\, x\}.$$

Der erste Schritt des Branch-and-Bound-Verfahrens besteht nun darin, durch Weglassen der Ganzzahligkeitsbedingung die LP-Relaxation zu lösen. Diese Lösung liefert uns, wegen obiger Bemerkung, eine obere Schranke (engl. bound) für das Problem. Im Anschluss wird das Problem nun solange „geschickt“ in disjunkte Teilprobleme verzweigt, bis dies nicht mehr möglich ist. Die LP-Relaxation vom Modell „KäMi“ hat die optimale Lösung:

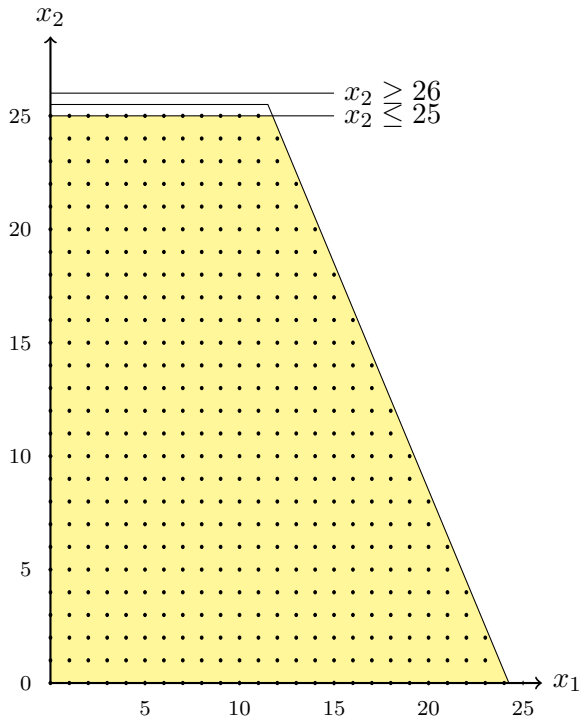



Abbildung 3.2: Die beiden disjunkten zulässigen Bereiche von dem Modell „KäMi“

$$\begin{aligned} x_1 &= 11.5 \\ x_2 &= 25.5 \\ f &= 29350. \end{aligned}$$

Dabei bezeichnet $f$ den Wert der Zielfunktion. Im Moment ist weder $x_1$, noch $x_2$ ganzzahlig. Der erste Iterationsschritt vom Branch-and-Bound-Verfahren verzweigt (engl. branch) nun das Problem und unterteilt es in zwei neue Teilprobleme. Betrachten wir etwa $x_2$ und machen uns bewusst, dass es zwischen 25 und 26 keine ganzen Zahlen gibt, können wir den zulässigen Bereich in zwei disjunkte Mengen aufteilen, indem wir die beiden Nebenbedingungen $x_2 \leq 25$ und $x_2 \geq 26$ hinzufügen. Wir erhalten die beiden Teilprobleme:

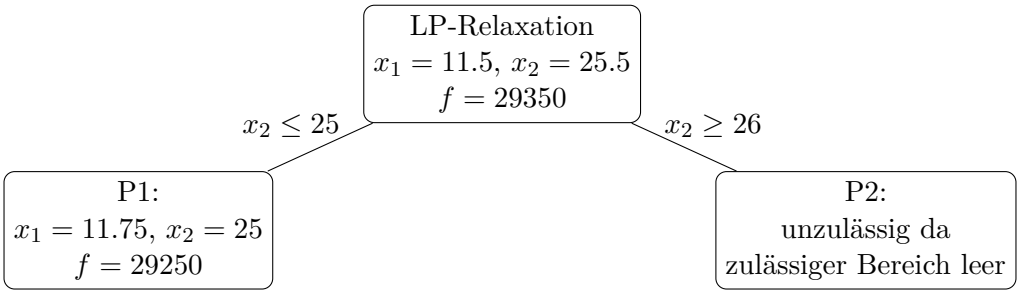
$$\begin{aligned} &\max 1000x_1 + 700x_2 \\ &\text{u.d.N. } 100x_1 + 50x_2 \leq 2425 \quad \text{(P1)} \\ &\qquad\qquad 20x_2 \leq 510 \\ &\qquad\qquad\quad x_2 \leq 25 \\ &\qquad x_1, x_2 \geq 0 \end{aligned}$$

$$\begin{aligned} &\max 1000x_1 + 700x_2 \\ &\text{u.d.N. } 100x_1 + 50x_2 \leq 2425 \quad \text{(P2)} \\ &\qquad\qquad 20x_2 \leq 510 \\ &\qquad\qquad x_2 \geq 26 \\ &\qquad x_1, x_2 \geq 0. \end{aligned}$$

In Abbildung 3.2 sieht man, wie die zulässigen Bereiche für die jeweiligen Probleme aussehen (gelb für P1, leer für P2). Der duale Simplex liefert nun für P1 die optimale Lösung:

$$\begin{aligned} x_1 &= 11.75 \\ x_2 &= 25 \\ f &= 29250. \end{aligned}$$

Für P2 erhalten wir keine Lösung, also ist der zulässige Bereich leer. Wir erhalten folgendes Baumdiagramm:




Das Procedere können wir nun mit P1 fortsetzen. Wir starten mit dem zweiten Iterationsschritt. Da $x_1$ nicht ganzzahlig ist, unterteilen wir auch hier wieder und erhalten die beiden neuen Teilprobleme:

$$\begin{aligned} &\max 1000x_1 + 700x_2 \\ &\text{u.d.N. } 100x_1 + 50x_2 \leq 2425 \quad \text{(P3)} \\ &\qquad\qquad 20x_2 \leq 510 \\ &\qquad\qquad x_2 \leq 25 \\ &\qquad\qquad x_1 \leq 11 \\ &\qquad x_1, x_2 \geq 0 \end{aligned}$$

$$\begin{aligned} &\max 1000x_1 + 700x_2 \\ &\text{u.d.N. } 100x_1 + 50x_2 \leq 2425 \quad \text{(P4)} \\ &\qquad\qquad 20x_2 \leq 510 \\ &\qquad\qquad x_2 \leq 25 \\ &\qquad\qquad x_1 \geq 12 \\ &\qquad x_1, x_2 \geq 0. \end{aligned}$$

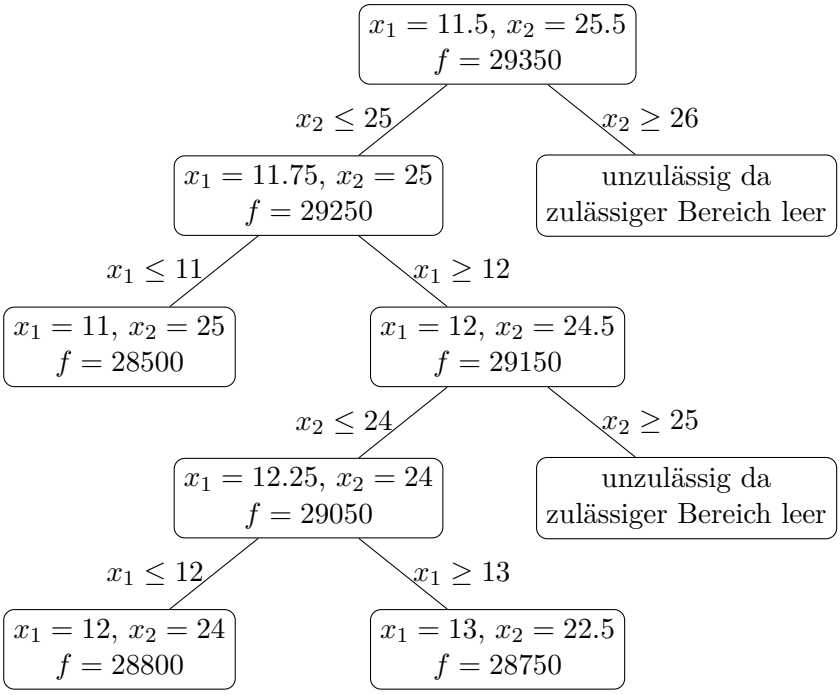



Abbildung 3.3: Fertiger Branch-and-Bound Baum

Wir erhalten mit dem dualen Simplex für P3

$$\begin{aligned} x_1 &= 11 \\ x_2 &= 25 \\ f &= 28500 \end{aligned}$$

und für P4

$$\begin{aligned} x_1 &= 12 \\ x_2 &= 24.5 \\ f &= 29150. \end{aligned}$$

Wir gehen also tiefer in den Baum, wodurch der zulässige Bereich immer kleiner wird. Dadurch wird der Wert der Zielfunktion auch immer kleiner. In der zweiten Iteration erreichen wir nun auch unsere erste ganzzahlige Lösung (und damit unseren ersten zulässigen Punkt), nämlich $x_1 = 11$, $x_2 = 25$. Deswegen müssen wir an diesem Punkt auch nicht mehr tiefer in den Baum. Der Zweig von P4 könnte aber durch weitere Verzweigung einen besseren Wert für die Zielfunktion liefern als der aktuelle (nämlich $f = 28500$). Deswegen müssen wir weiter iterieren und verzweigen mit $x_2 \leq 24$ und $x_2 \geq 25$. Führen wir dies soweit wie möglich aus, erhalten wir den sogenannten Branch-and-Bound-Baum wie in Abbildung 3.3.
Es ist nicht mehr notwendig, bei P8 mit $x_1 = 13$, $x_2 = 22.5$ den Baum weiter zu verzweigen, da diese optimale Lösung kleiner ist ($f = 28750$), als unsere bis dahin beste Lösung mit $f = 28800$. Somit hat das Branch-and-Bound-Verfahren nach vier Iterationen die optimale Lösung unseres Problems gefunden. Die optimale Lösung lautet:

$$
\begin{aligned}
f^* &= 28000 \\
x_1^* &= 12, \\
x_2^* &= 24.
\end{aligned}
$$

Was wir gesehen haben ist zum einen, dass Runden nicht unbedingt zur besten Lösung führt. Zum zweiten haben wir gesehen, dass wir mit dem Branch-and-Bound-Verfahren nicht den gesamten Lösungsraum enumerieren müssen.

## 3.2 Die allgemeine Form des Branch-and-Bound

Im Folgenden werden wir das Branch-and-Bound-Verfahren für allgemeine ganzzahlige lineare Optimierungsprobleme beschreiben (ILP, engl. Integer Linear Programming). Dieses Verfahren lässt sich sehr einfach auf gemischt ganzzahlige lineare Optimierungsprobleme (MILP, engl. Mixed Integer Linear Programming) verallgemeinern. Dabei beschreiben wir das Verfahren ausschließlich für Maximierungsprobleme. Für ILP, die zu minimieren sind, verweisen wir auf die Bemerkungen 2.1.2 und 3.2.3. Ein MILP hat stets die folgende Form:

$$
\begin{aligned}
&\max c_1^T x_1 + c_2^T x_2 \\
\text{u.d.N. } &A_1 x_1 + A_2 x_2 \leq b \\
&x_1 \geq 0 \\
&x_2 \geq 0, \text{ und ganzzahlig}
\end{aligned}
$$

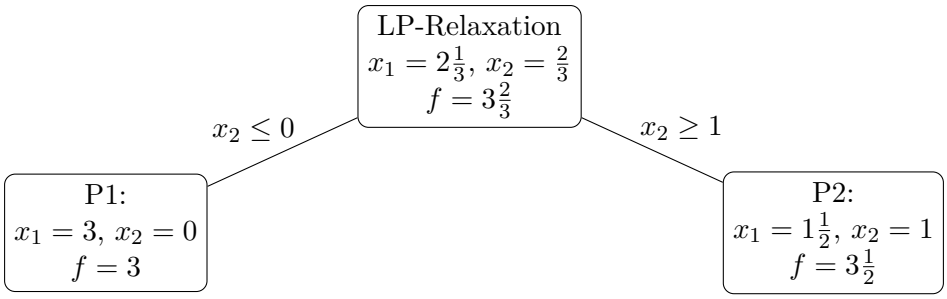
mit $c_1 \in \mathbb{R}^{n_1}$, $c_2 \in \mathbb{R}^{n_2}$, $A_1 \in \mathbb{R}^{m \times n_1}$, $A_2 \in \mathbb{R}^{m \times n_2}$, und $b \in \mathbb{R}^m$. Das Branch-and-Bound-Verfahren für ILP kann dann sehr einfach auf MILP verallgemeinert werden. Dafür genügt es, das Verfahren nur auf $x_2$ anzuwenden. Dazu betrachten wir noch einmal ein kleines Beispiel:

**Beispiel 3.2.1** Betrachte das folgende MILP:

$$
\begin{aligned}
&\max x_1 + x_2 \\
\text{u.d.N. } &x_1 + x_2 \leq 3 \\
&2x_1 + 5x_2 \leq 8 \\
&x_1, x_2 \geq 0, \; x_2 \text{ ganzzahlig.}
\end{aligned}
$$

Gesucht ist nun eine optimale Lösung, wobei nur $x_2$ ganzzahlig sein muss. Zunächst lösen wir die LP-Relaxation und erhalten als optimale Lösung $x_1 = 2\frac{1}{3}$, $x_2 = \frac{2}{3}$ und $f = 3\frac{2}{3}$. Da $x_2$ die einzige Variable ist, die die Ganzzahligkeitsbedingung erfüllen muss, müssen wir $x_2$ verzweigen. Wir erhalten also die Teilprobleme mit $x_2 \leq 0$ (das bedeutet, dass gilt $x_2 = 0$), und $x_2 \geq 1$.
Die optimale Lösung des ersten Teilproblems (mit $x_2 \leq 0$) liefert die Lösung $x_1 = 3$, $x_2 = 0$, $f = 3$, die optimale Lösung des zweiten Teilproblems (mit $x_2 \geq 1$) liefert $x_1 = 1\frac{1}{2}$, $x_2 = 1$, $f = 3\frac{1}{2}$. Die Lösung des ersten Teilproblems muss nicht weiter verzweigt werden, da der Wert der Zielfunktion hier kleiner ist als die Lösung beim zweiten Teilproblem.

Da die Lösung des zweiten Teilproblems ebenfalls zulässig ist, müssen wir hier auch nicht weiter verzweigen und sind bereits fertig. Die optimale Lösung lautet daher $x_1^* = 1\frac{1}{2}$, $x_2^* = 1$ und $f^* = 3\frac{1}{2}$. Der vollständige Branch-and-Bound Baum sieht wie folgt aus:




### 3.2.1 Das Verfahren für ILP

Wie wir in dem Beispiel zu „KäMi“ bereits gesehen haben, beruht das Branch-and-Bound-Verfahren auf zwei Lösungsprinzipien; das Branching (Verzweigen) und das Bounding (Beschränken). Gegeben sei ein ILP mit

$$\begin{aligned} &\max c^T x \\ \text{u.d.N. } &Ax \leq b \quad (P_0) \\ &x \geq 0, \text{ ganzzahlig} \end{aligned}$$

sowie $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ und $b \in \mathbb{R}^m$.

**1. Branching:** Das zu lösende Problem $P_0$ wird so in $k$ Teilprobleme zerlegt, dass

1. $\mathrm{K}(P_0) = \bigcup_{i=1}^{k} \mathrm{K}(P_i)$ und möglichst
2. $\mathrm{K}(P_i) \cap \mathrm{K}(P_j) = \emptyset$ für alle $i \neq j$ gilt.

Dabei bezeichnet $\mathrm{K}(P_i)$ den jeweiligen zulässigen Bereich. Die Probleme $P_1, \ldots, P_k$ sind analog zu $P_0$ weiter zu verzweigen. Somit erhalten wir einen (Lösungs-)Baum von Problemen, wie er in Abbildung 3.3 dargestellt wurde. Das jeweilige Ausgangsproblem bezeichnet man mit der Wurzel des Baumes bzw. der Wurzel des Teilproblems.

**2. Bounding:** Hier werden Schranken für die Werte der Zielfunktion ermittelt und Teilprobleme ausgelotet. Eine globale untere Schranke kann immer mit $f_u := -\infty$ angegeben werden. Im Allgemeinen kann aber, z.B. durch Anwendung einer Heuristik, eine bessere untere Schranke $f_u$ angegeben werden. Die aktuell beste und bekannte zulässige Lösung, die im Branch-and-Bound-Verfahren gefunden wurde, liefert dann die aktuell beste untere Schranke.

Weiter kann für jedes Teilproblem $P_i$, $(i = 1, 2, \ldots)$ eine lokale obere Schranke $f_i^o$ für den Wert der Zielfunktion von $P_i$ ermittelt werden. Dies wird wie im Beispiel gesehen, über die LP-Relaxation $P_i'$ von $P_i$ gelöst, d.h. wir verzichten auf die Ganzzahligkeitsbedingung und lösen das LP mit dem Simplex-Verfahren. Damit erhalten wir $\mathrm{K}(P_i) \subset \mathrm{K}(P_i')$ und eine lokale obere Schranke (vgl. Bemerkung 3.1.1).

Ein Teilproblem $P_i$ heißt ausgelotet, falls es eines der drei sich ausschließenden Bedingungen erfüllt:

1. ($f_i^0 \leq f_u$): Die optimale Lösung des Teilproblems kann nicht besser sein als die beste bekannte zulässige Lösung.

2. ($f_i^o > f_u$ und die optimale Lösung von $P_i'$ ist zulässig für $P_i$ und damit auch für $P_0$): Hier wurde eine neue beste Lösung für das Problem $P_0$ gefunden, diese ist zu speichern und man setzt $f_u := f_i^o$.

3. (K($P_i'$)=$\emptyset$): $P_i'$ besitzt keine zulässige Lösung; damit gilt auch K($P_i$)=$\emptyset$

Ist $P_i$ ausgelotet, so muss dieses Teilproblem nicht weiter betrachtet und auch nicht weiter verzweigt werden.

**Definition 3.2.2** Die Abrundungsfunktion (oder untere Gaußklammer oder floor Funktion) ist eine Funktion, die einer reellen Zahl $x \in \mathbb{R}$ die nächst kleinere ganze Zahl zuordnet. Wir schreiben hierfür $\lfloor x \rfloor$

Damit können wir den folgenden Branch-and-Bound Algorithmus für ILP vorstellen.

**Algorithmus :** Branch-and-Bound Algorithmus für ILP

| | |
|---|---|
| **Input** | : Ein ILP P von der Form $\max c^T x$ u.d.N. $Ax \leq b$, $x \geq 0$ ganzzahlig mit $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ und $b \in \mathbb{R}^m$ |
| **Output** | : Entweder:<br>(i) Das Modell hat keine ganzzahlige Lösung, oder<br>(ii) eine optimale Lösung für das ILP |
| **1. Schritt** | : *Initialisieren.* Setze $P_0 := P$, $K_0 := K(P_0)$, Index:= $\{0\}$, $f_u$;= $-\infty$ und $t := 0$ |
| **2. Schritt** | : *Auswahl.* Wähle ein $k \in$ Index und gehe zum 3. Schritt |
| **3. Schritt** | : *Bounding.* Berechne die optimale Lösung $(f_k^0, x^k)$ der LP-Relaxation von $P_k$ (sofern existent). Wenn es keine gibt, setze $f_k = -\infty$. Gehe zum 4. Schritt |
| **4. Schritt** | : *Ausloten.* $P_k$ heißt ausgelotet, d.h. braucht nicht weiter verzweigt zu werden, wenn einer der drei Fälle eintritt:<br>(a) $f_k^o \leq f_u$<br>(b) $f_k^o > f_u$ und $x^k$ ganzzahlig<br>(c) $K_k = \emptyset$<br>Wenn $P_k$ ausgelotet, setze Index := Index $\setminus\{k\}$ und gehe zum 6. Schritt. Erfüllt $P_k$ zusätzlich noch den Auslotungsfall (b), setze $f_u := f_k^o$. Sonst gehe zum 5. Schritt |
| **5. Schritt** | : *Branching.* Zerlege $K_k$ wie folgt; wähle $x_i^k$ $(i = 1, \ldots, n)$ mit $x_i^k \notin \mathbb{Z}$, setze $K_{t+1} := K_k \cup \{x_i \leq \lfloor x_i^k \rfloor\}$ und $K_{t+2} := K_k \cup \{x_i \geq \lfloor x_i^k \rfloor + 1\}$ sowie $P_{t+1} := \max\{c^T x | x \in K_{t+1}\}$, $P_{t+2} := \max\{c^T x | x \in K_{t+2}\}$, $t := t + 2$, Index:=Index$\cup\{t+1, t+2\}$ und gehe zum 2. Schritt. |
| **6. Schritt** | : *Optimalitätstest und Abbruchkriterium.* Wenn Index $\neq \emptyset$ gehe zum 2. Schritt. Stop wenn Index = $\emptyset$, die aktuell beste Lösung ist die optimale Lösung, wenn es keine gibt, d.h. wenn $f_u = -\infty$, dann gibt es keine Optimale Lösung |

Dabei können wir $P_0$ mit dem vorgestellten Simplex-Verfahren lösen, während wir sämtliche $P_k$ für $k > 0$ mit dem dualen Simplex lösen können.

**Bemerkung 3.2.3** Für ILP's, die zu minimieren sind, kann obiger Algorithmus einfach angepasst werden. Dazu genügt es, $f_u$ durch die aktuelle obere Grenze $f_k^o$ und $-\infty$ durch $\infty$ zu ersetzen, sowie die Ungleichheitszeichen umzudrehen. [9]

**Satz 3.2.4** Gegeben sei ein ILP mit

$$\begin{aligned} &\max c^T x \\ \text{u.d.N. } &Ax \leq b \quad (P) \\ &x \geq 0, \text{ ganzzahlig} \end{aligned}$$

sowie $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ und $b \in \mathbb{R}^m$. Dabei sei der zulässige Bereich $\mathrm{K}(P)$ abgeschlossen und rational. Dann gilt:

- Ist $\max\{c^T x | \, x \in \mathrm{K}(P)\}$ unzulässig oder unbeschränkt, so bricht das Verfahren ab und zeigt an, dass $(P)$ keine Lösung besitzt
- Hat $\max\{c^T x | \, x \in \mathrm{K}(P)\}$ ein endliches Optimum, und ist $P_0 \neq \emptyset$, so terminiert der obige Algorithmus mit einer Optimallösung von $(P)$
- Hat $\max\{c^T x | \, x \in \mathrm{K}(P)\}$ ein endliches Optimum, und ist $P_0 = \emptyset$, so kann durch Einführen einer unteren Schranke für den Zielfunktionswert ein endlicher Abbruch erzwungen werden.

*Beweis:* Das Verfahren enumeriert implizit alle ganzzahligen Lösungen von $\mathrm{K}(P)$.[10] □

### 3.2.2 Freiheiten bei der Implementierung

Wie in dem obigen Algorithmus leicht zu sehen ist, gibt es natürlich noch einige Freiheiten bei der Implementierung. Die hier vorgestellte Übersicht orientiert sich an [Dom15], Kapitel 6.4. Eine tiefer gehende Betrachtung, insbesondere die für die Laufzeit entscheidenden[11] Punkte 2 *Auswahl* und 4 *Ausloten* bzw. 5 *Branching*, findet sich in [Mar99] Kapitel 9, sowie [Hoo12] Kapitel 5 und 7.

1. Bereits im ersten Schritt gibt es die Möglichkeit, eine möglicherweise bessere untere Schranke $f_u$ anzugeben. Hierbei wird eine Heuristik verwendet. Da wir hier nicht weiter auf dieses Thema eingehen, verweisen wir auf [Wol80].
2. Im zweiten Iterationsschritt des Algorithmus sind vor allem zwei grundsätzliche Auswahlregeln zu nennen:
   - LIFO-(Last in - First Out) Regel: Hier wird das jeweils zuletzt in die Kandidatenliste aufgenommene Teilproblem weiter bearbeitet (depth first search). Diese wird nochmals in zwei mögliche Varianten unterteilt:

[9] Vergleiche hierzu [Sie96] Kapitel 3.2.2

[10] Dieser Beweis ist in [Hel14], Lemma 8.7, zu finden.

[11] In [Mar99] Kapitel 9.5 heißt es dazu: „Two decisions which greatly affect the effectiveness of a linear programming based branch-and-bound algorithm are node selection ... and branching variable selection“

(i) Reine Tiefensuche (laser search): Hier wird für jedes Problem zunächst nur ein Teilproblem gebildet und das ursprüngliche Problem wieder in die Kandidatenliste aufgenommen.

(ii) Tiefensuche mit vollständiger Verzweigung: Hier wird jedes Problem vollständig verzweigt und anschließend aus der Kandidatenliste wieder entfernt. Eines der Teilprobleme wird dann weiter verarbeitet während die übrigen weiter in der Kandidatenliste bleiben. Bei der Reihenfolge der Teilprobleme kann wieder eine zusätzliche Auswahlregel angewandt werden (z.B. MUB).

- MUB-(Maximum Upper Bound) Regel: Nach dieser Regel wird stets jenes Problem $P_k$ mit der größten oberen Schranke $f_k^o$ gewählt, in der Hoffnung, dass die oder eine optimale Lösung von $P_0$ am ehesten unter den zulässigen Lösungen von $P_k$ befindet. Es handelt sich um eine Breitensuche (breadth first search). Mögliche Probleme gegenüber der LIFO-Regel können sich bei der Kandidatenliste ergeben. Diese kann hier sehr groß werden (Speicherplatzbedarf). Dagegen ist die erste gefundene zulässige Lösung in der Regel sehr gut.

Nach Wolsey[12] wird in der Praxis häufig ein Kompromiss aus beiden Verfahren verwendet. Zunächst wird mit einem Initial-Verfahren mit der LIFO-Regel eine zulässige Lösung gesucht, um im Anschluss mit der MUB-Regel die optimale Lösung zu finden. Dies geschieht in der Hoffnung, schnell eine bessere untere Schranke zu bekommen, um den Lösungsbaum nicht so weit verzweigen zu müssen.

3. Da das Branch-and-Bound-Verfahren nicht nur für ILP (bzw. MILP) angewandt werden kann, sei zum dritten Schritt noch erwähnt, dass es neben der LP-Relaxation noch andere Relaxationsmöglichkeiten gibt. Zu erwähnen sind hier etwa die Lagrange-Relaxation, Weglassen von Nebenbedingungen und die Surrogate-Relaxation. Näheres dazu findet sich in [Dom15], Kapitel 6.4.

4. Neben den im vierten Schritt vorgeschlagenen Auslotungsfällen, kann auch mit Hilfe von sogenannten logischen Tests die Kandidatenliste verkleinert werden. Diese können zu einer Verbesserung der Schranken führen und damit zum Ausloten der Kandidaten. Weiter können Schranken verschärft werden, indem etwa Restriktionen (Schnitte, Cuts) hinzugefügt werden. Diese Vorgehensweise führt dann zum Branch-and-Cut-Verfahren. Hier verweisen wir auf [Kor12], Kapitel 21.

5. Im fünften Schritt des vorgestellten Algorithmus kann neben dem zu wählenden $i \in \{1, \ldots, n\}$ auch die Zerlegung in Teilprobleme verallgemeinert werden. Die Art und Weise der Zerlegung von $P_k$ in Teilprobleme hängt wesentlich von dem gegebenen ILP ab. Die Auswahl von $i$ kann etwa so erfolgen, dass der kleinste (oder größte) mögliche Index gewählt wird und das Problem dann, wie hier vorgestellt, in genau zwei Teilprobleme zerlegt wird. Möglich wäre auch, jenes $i$ so zu wählen, dass $x_i^k$ am weitesten von der Ganzzahligkeit entfernt ist. Alternativ können aber alle (oder ein Teil) der möglichen Kandidaten überprüft werden, um dann den zu nehmen, der die größte Wertminderung der Zielfunktion verspricht (strong branching). Weiterführendes über die Auswahl von $i$ ist in [Bel13] zu finden.

[12] vgl. [Wol98] Kapitel 7.4

6. Alternative Abbruchkriterien könnten sein, dass man sich mit einer hinreichend guten, oder sogar der ersten zulässigen (d.h. ganzzahligen) Lösung zufrieden gibt.

Zu bemerken bleibt, dass es kein allgemeingültiges Verfahren gibt, das für jedes ILP besonders gut bzw schnell funktioniert. Ein Beispiel hierzu, welches das Rucksackproblem betrachtet, findet sich in [Sch97]. Einen Vergleich, wie sich das Aufteilen in Teilprobleme auf die Laufzeit von Branch-and-Bound auswirkt, ist in [Ach05] zu finden.

## 3.3 Anwendungen der ganzzahligen Optimierung

In diesem Abschnitt wollen wir einige Anwendungen von linearer ganzzahliger Optimierung sowie einige klassische Beispiele aus der kombinatorischen Optimierung betrachten. Dieser Abschnitt orientiert sich an [Sie96], Abschnitte 3.2.3 und 3.2.4, sowie [SuMe13], fünftes Kapitel.

### 3.3.1 Anwendungen

Im Vergleich zur linearen Programmierung, erlaubt die Ganzzahligkeitsbedingung wesentlich mehr Spielraum bei der Modellierung. Im Wesentlichen liegt dies an folgenden Gründen:

1. Die gesuchten Größen müssen ganzzahlig sein, da es keinen Sinn ergäbe, z.B. 4.5 Busse zu bauen.

2. Die gesuchten Größen sind sogar auf die Binärvariablen 0 und 1 beschränkt. Dies kommt bei Entscheidungsproblemen zum Tragen. So macht es etwa keinen Sinn, einen optimalen Busfahrplan zu erstellen, bei dem ein Bus zu 0.6 fährt. Entweder er fährt (= 1), oder er fährt nicht (= 0).

Zwei typische Anwendungen sind:

**Öffentlicher Personennahverkehr:**
Ein Beispiel wäre hier die Erstellung eines Dienstplans für ein Busnetz in einer Stadt. Dabei ist zu beachten, dass die geplanten Linien alle mit Fahrern besetzt werden müssen. Weiter müssen die Fahrer gesetzliche Pausen einhalten, außerdem müssen die Busse natürlich pünktlich nach Fahrplan fahren. Hier kann etwa mit Binärvariablen modelliert werden, wobei eine 1 bedeutet, dass der Fahrer die Linie fährt und eine 0 bedeutet, das der Fahrer die Linie nicht fährt.

**Produktionsplanung:**
Typische Beispiele für die Produktionsplanung sind etwa Vorgänge in der Industrie, bei denen Material, Arbeitszeit, Maschinenzeit etc., in gewissen Grenzen vorhanden sind und für die Produktion von bestimmten Produkten benötigt werden. Ziel ist meistens ein Produktionsplan, der einen größtmöglichen Gewinn garantiert. Aber auch eine möglichst hohe Auslastung kann gesucht sein, oder möglichst geringe Kosten.

### 3.3.2 Rucksackproblem

Das Rucksackproblem kann normalerweise effizient mit dem Branch-and-Bound-Verfahren gelöst werden. Der Name des Problems leitet sich von der folgenden Interpretation ab.

Eine gewisse Anzahl an Objekten soll in einen Rucksack gepackt werden. Dabei hat jedes dieser Objekte ein gewisses Gewicht (oder ein gewisses Volumen) und ein bestimmtes Maß an Nutzen. Der Rucksack soll nun so gepackt werden, dass der Nutzen möglichst groß ist. Gesucht ist also die Kombination von Objekten, die dies erfüllt. Dieses Problem wird häufig in den Anwendungen betrachtet. So kann z.B. ein LKW-Fahrer nur eine begrenzte Menge an Ware transportieren. Er möchte nun soviel Ware wie möglich mitnehmen, aber so, dass der Gegenwert der Ware in seinem LKW, möglichst hoch ist.

Betrachten wir ein Beispiel im Fall von fünf Objekten, die wie folgt dargestellt durch ihren jeweiligen Nutzen und ihr jeweiliges Volumen, repräsentiert sind:

| Objekt | Nutzen ($\times$ 100 €) | Volumen ($\times$ 1 Liter) |
|---|---|---|
| 1 | 5 | 5 |
| 2 | 3 | 4 |
| 3 | 6 | 7 |
| 4 | 6 | 6 |
| 5 | 2 | 2 |

Der Rucksack habe ein Volumen von 15 Liter. Die beste Wahl für dieses Rucksackproblem ist dann die Lösung des ILP:

$$\begin{aligned} \max\ & 5x_1 + 3x_2 + 6x_3 + 6x_4 + 2x_5 \\ \text{u.d.N}\ & 5x_1 + 4x_2 + 7x_3 + 6x_4 + 2x_5 \leq 15 \\ & x_1, x_2, x_3, x_4, x_5 \in \{0, 1\} \end{aligned}$$

Diese Sorte von Rucksackproblemen lässt sich sehr gut mit dem Branch-and-Bound-Verfahren lösen. Dies liegt vor allem an zwei Aspekten. Erstens, da jede Variable 0 oder 1 sein muss, gibt es nur zwei Verzweigungsfälle für $x_i$, nämlich $x_i = 0$ oder $x_i = 1$. Zweitens, können die LP-Relaxationen durch einfaches Überprüfen gelöst werden. Dazu stellen wir fest, dass $c_i/a_i$ der Nutzen von Objekt $i$ für jede Einheit des Rucksackes ist. Daher haben die besten Objekte auch den größten Wert von $c_i/a_i$, die schlechtesten den niedrigsten. Um ein Teilproblem zu lösen, müssen die Brüche $c_i/a_i$ berechnet und nach Größe sortiert werden (von groß nach klein). Das Verfahren beginnt nun mit dem bestplatzierten Objekt, dann mit dem zweitbesten usw., bis das nächstbeste Objekt den Platz im Rucksack überschreiten würde. Etwas präziser ist damit also der folgende Satz zu zeigen:

**Satz 3.3.1** Die LP-Relaxation

$$\begin{aligned} \max\ & c_1x_1 + \cdots + c_nx_n \\ \text{u.d.N.}\ & a_1x_1 + \cdots + a_nx_n \leq b \qquad P \\ & 0 \leq x_i \leq 1 \text{ für } i = 1, \ldots, n \end{aligned}$$

hat die optimale Lösung:

$$\begin{aligned} x_1^* = \ldots x_r^* &= 1, \\ x_{r+1}^* &= \frac{1}{a_{r+1}}(b - a_1 - \ldots - a_r), \\ x_{r+2}^* = \ldots = x_n^* &= 0, \end{aligned}$$

mit $a_1 + \ldots a_n \leq b$ und $a_1 + \ldots a_{r+1} > b$ und $c_1/a_1 \geq \ldots \geq c_n/a_n$. Das heißt, die ersten $r$ Objekte lassen noch genügend Platz im Rucksack, das $r+1$-te Objekte würden ihn aber überfüllen.

*Beweis:* Es ist klar, dass $0 \leq \frac{1}{a_{r+1}}(b - a_1 - \ldots - a_r) < 1$ gilt. Weiter ist klar, dass $x^* = (x_1^*, \ldots x_n^*)$ zulässig ist. Zu zeigen bleibt also, das $x^*$ optimal ist.

Um das duale Problem zu bestimmen, schreiben wir $P$ in Normalform I:

$$\begin{aligned} &\max c^T x \\ &\text{u.d.N. } Ax \leq \tilde{b} \\ &\qquad x \geq 0 \end{aligned}$$

mit:

$$c = \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} \quad A = \begin{pmatrix} a_1 & \cdots & a_n \\ 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{pmatrix} \quad \text{und} \quad \tilde{b} = \begin{pmatrix} b \\ 1 \\ \vdots \\ 1 \end{pmatrix}$$

Damit ist das duale Problem zu $P$:

$$\min \begin{pmatrix} b & 1 & \cdots & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ \vdots \\ y_{n+1} \end{pmatrix}$$

$$\text{u.d.N} \begin{pmatrix} a_1 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ a_n & 0 & \cdots & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ \vdots \\ y_{n+1} \end{pmatrix} \geq \begin{pmatrix} c_1 \\ \vdots \\ c_{n+1} \end{pmatrix}$$

$$y \geq 0$$

bzw.

$$\min b y_1 + y_2 + \cdots + y_{n+1}$$

$$\begin{aligned} &\text{u.d.N. } a_1 y_1 + y_2 \geq c_1 \\ &\qquad \vdots \\ &\qquad a_n y_1 + y_{m+1} \geq c_n \end{aligned}$$

$$y_1, \ldots y_{n+1} \geq 0.$$

Wir setzen

$$
\begin{aligned}
y_1^* &= \frac{c_{r+1}}{a_{r+1}}, \\
y_k^* &= c_{k-1} - a_{k-1}\left(\frac{c_{r+1}}{a_{r+1}}\right) \text{ für } k = 2, \dots r+1 \text{ und} \\
y_k^* &= 0 \text{ für } k = r+2, \dots n+1
\end{aligned}
$$

Dann ist $y^* = \left(y_1^* \dots y_{n+1}^*\right)^T$ dual zulässig und wir sehen, dass

$$c^T x^* = \tilde{b}^T y^*$$

ist. Nach Korollar 2.2.3 zum schwachen Dualitätssatz gilt daher, dass $x^*$ eine optimale Lösung für $P$ ist. □

Kehren wir nochmals zu obigem Beispiel zurück, um eine Anwendung des Verfahren zu betrachten. Wir starten mit der Berechnung der Brüche $c_i/a_i$ und fügen ein Ranking ein:

| Objekt | $c_i/a_i$ | Ranking (1 = am besten, 5 = am schlechtesten) |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 0.75 | 5 |
| 3 | 0.86 | 4 |
| 4 | 1 | 1 |
| 5 | 1 | 1 |

Die LP-Relaxation kann nun wie folgt gelöst werden. Es gibt drei Objekte mit dem besten Ranking. Wir wählen Objekt 1 sodass $x_1 = 1$ ist. Dann sind noch $(b - a_1 x_1 = 15 - 5 =) 10$ Liter frei. Als nächstes wählen wir das zweitbeste Objekt (Entweder 4 oder 5). Wir wählen $x_4$ (d.h. $x_4 = 1$). Nun sind noch $(b - a_1 x_1 - a_4 x_4 =)$ 4 Liter frei. Jetzt wählen wir Objekt 5 ($x_5 = 1$) womit noch 2 Liter Platz verbleiben. Das nächstbeste Objekt ist 3. Wir füllen den Rucksack so gut es geht mit Objekt 3. Da nur noch 2 Liter Platz vorhanden sind, wählen wir $x_3 = \frac{2}{7}$, was bedeutet, dass nur $\frac{2}{7}$ von Objekt 3 in den Rucksack gehen. Nun ist der Rucksack voll. Die zulässige Lösung der LP-Relaxation ist also:

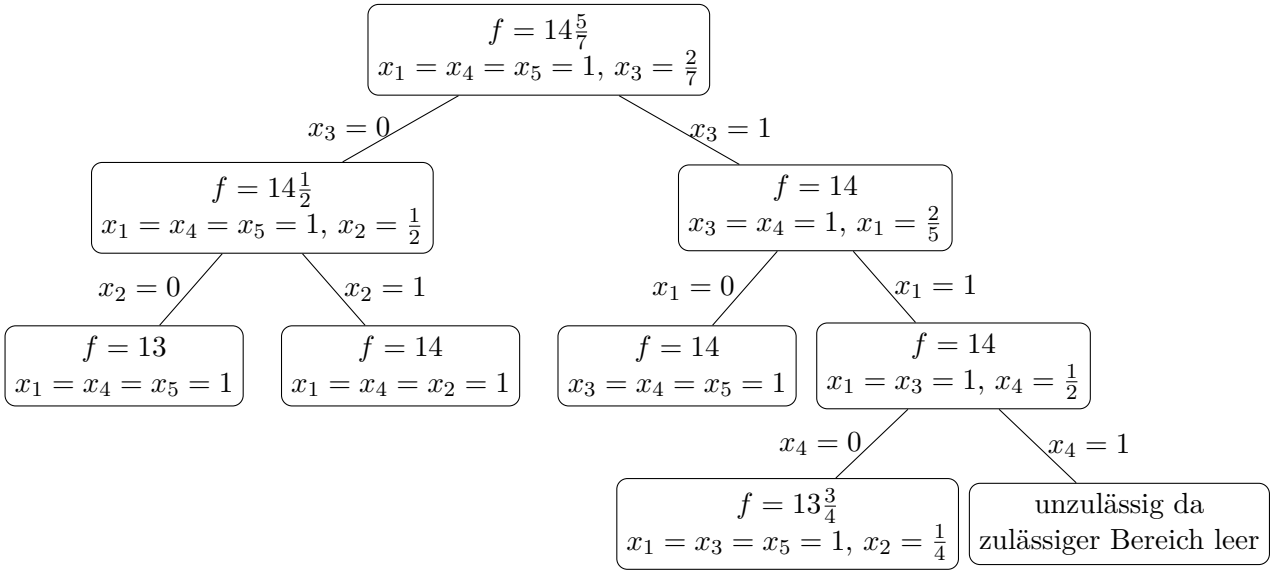
$$
\begin{aligned}
&x_1 = 1, \quad x_2 = 0, \quad x_3 = \frac{2}{7}, \quad x_4 = 1, \quad x_5 = 1 \\
&\text{mit } f = 14.72
\end{aligned}
$$

Um zu zeigen, dass diese Lösung bereits optimal ist, berechnen wir $y$ wie in obigem Satz. Es gilt

$$
\begin{aligned}
y_1 &= c_3/a_3 = 0.86 \\
y_2 &= c_1 - a_1(c_3/a_3) = 0.7 \\
y_5 &= c_4 - a_4(c_3/a_3) = 0.84 \\
y_6 &= c_5 - a_5(c_3/a_3) = 0.28 \\
y_3 &= y_4 = 0
\end{aligned}
$$

da $c_1/a_1 = c_4/a_4 = c_5/a_5 \geq c_3/a_3 \geq c_2/a_2$. Weiter gilt $g = by_1 + y_2 + y_3 + y_4 + y_5 + y_6 = 14.72$. Damit ist $x = (1,\ 0,\ \frac{2}{7},\ 1,\ 1)^T$ optimal.

Dieser Prozess wird nun solange fortgeführt, bis eine optimale Lösung für das gesuchte ILP gefunden wurde. Der fertige Branch-and-Bound Baum sieht wie folgt aus (Null-Werte sind nicht eingetragen):




Das Branch-and-Bound-Verfahren hat zwei optimale Lösungen gefunden:

$$x_1^* = 1,\ x_2^* = 1,\ x_3^* = 0,\ x_4^* = 1,\ x_5^* = 0 \text{ mit } f^* = 14, \text{und}$$
$$x_1^* = 0,\ x_2^* = 0,\ x_3^* = 1,\ x_4^* = 1,\ x_5^* = 1 \text{ mit } f^* = 14$$

Wir sehen auch, dass eine Einheit im Rucksack übrig bleibt.

Abschließend hier das allgemeine Rucksackproblem:

**Definition 3.3.2** Das allgemeine Rucksackproblem wird beschrieben durch

| | ILP Modell des Rucksackproblems |
|---|---|
| max | $c_1x_1 + \cdots + c_nx_n$ |
| u.d.N. | $a_1x_1 + \cdots + a_nx_n \leq b$ |
| | $x_1, \cdots, x_n \geq 0,$ und ganzzahlig |

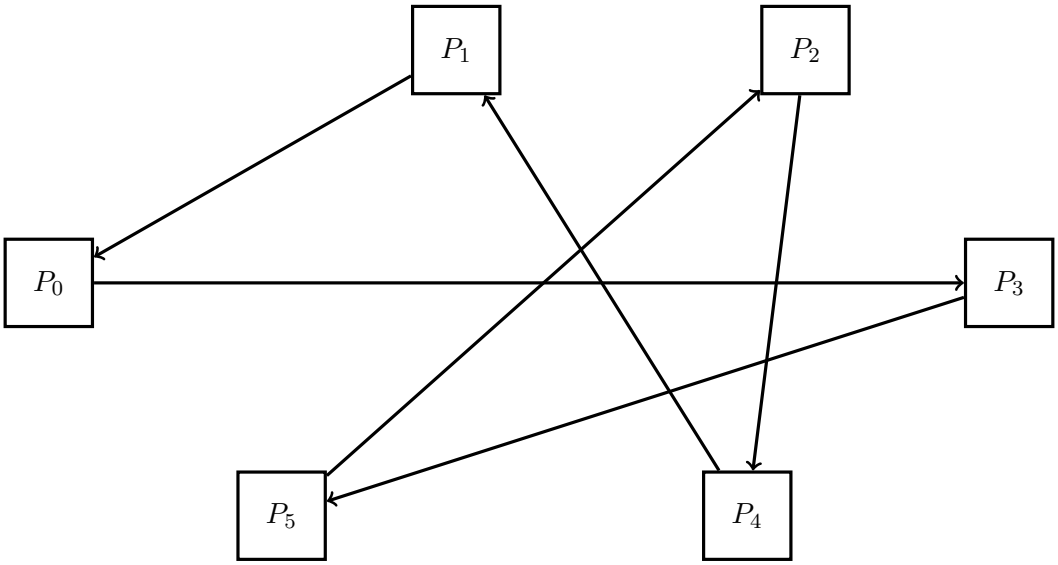
mit $n$ Objekten, $c_i\,(> 0)$ der Nutzen des Objekts $i$, $b\,(> 0)$ das Volumen des Rucksackes und $a_i\,(> 0)$ das Gewicht/Kosten/Volumen von Objekt $i$.

### 3.3.3 Maschinenplanungs-Problem

Das Problem aus diesem Abschnitt benutzt das Branch-and-Bound-Verfahren so, dass zwei oder mehr Teilprobleme gebildet und gelöst werden. Das Problem kann wie folgt

formuliert werden: Eine Maschine hat $n$ ($\geq 2$) Aufgaben zu erledigen. Die Bearbeitungszeit von Aufgabe $i$ beträgt $p_i$ (Minuten). Nach jeder Aufgabe muss die Maschine für die nächste Aufgabe neu justiert werden. Die Zeit für die neue Justierung von Aufgabe $i$ zu Aufgabe $j$ sei $c_{ij}$ (Minuten). Nachdem alle Aufgaben erledigt sind, muss die Maschine in ihre Ausgangsposition (Ausgangseinstellung) zurückgesetzt werden. Die Einrichtungszeit von der Ausgangsposition zu Aufgabe $i$ ist $c_{0i}$ (Minuten), und für Aufgabe $j$ zur Ausgangsposition ist sie $c_{j0}$ (Minuten). Das Problem besteht nun darin, die Aufgaben so zu verteilen, dass die Bearbeitungszeit aller Aufgaben minimal ist. Da alle Aufgaben von einer Maschine verarbeitet werden, hängt die Bearbeitungszeit nur von den Einrichtungszeiten ab. Also muss die Summe der Einrichtungszeiten minimiert werden.

Als Beispiel nehmen wir an, wir haben fünf Aufgaben $P_1, P_2, P_3, P_4$ und $P_5$ zu erledigen (also $n = 5$). Eine mögliche Verteilung sieht wie folgt aus:




Dabei zeigen die Pfeile an, in welcher Reihenfolge die Aufgaben erledigt werden und $P_0$ sei die Ausgangsposition. Wir führen die folgenden Binärvariablen ein. Für alle $i, j = 0, \cdots, n$ sei:

$$\delta_{ij} = \begin{cases} 1 & \text{Aufgabe } j \text{ wird nach Aufgabe } i \text{ ausgeführt} \\ 0 & \text{sonst} \end{cases}$$

Damit erhalten wir die Zielfunktion

$$\min \sum_{i=0}^{n} \sum_{j=0}^{n} \delta_{ij} c_{ij}.$$

Die Nebenbedingungen können ebenfalls mit $\delta_{ij}$ ausgedrückt werden. Da jede Aufgabe genau einmal ausgeführt werden soll erhalten wir

$$\sum_{i=0}^{n} \delta_{ij} = 1 \qquad \text{für } j = 0, \cdots, n$$

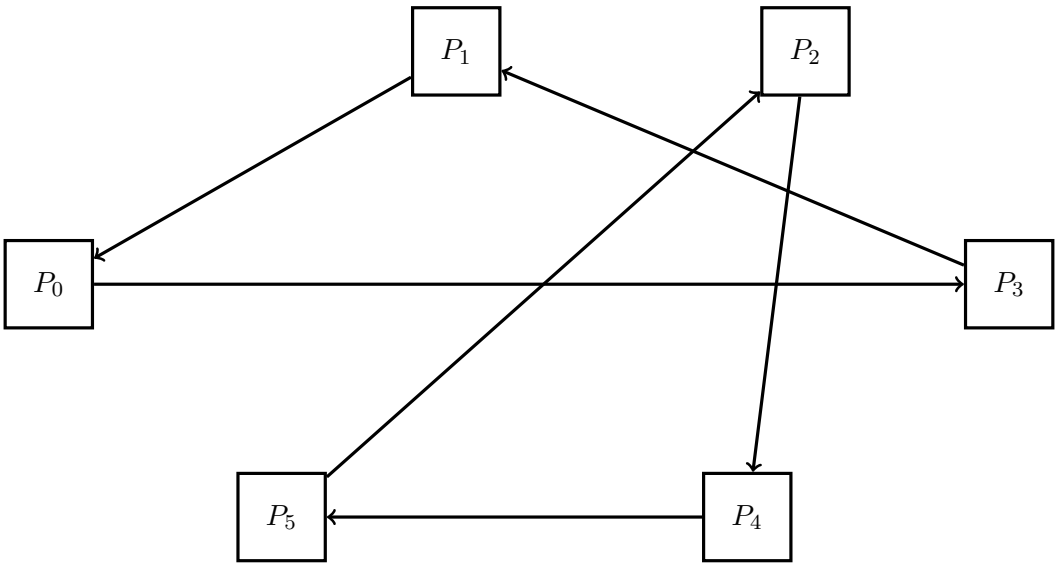
und da nach jeder Aufgabe eine andere Aufgabe ausgeführt werden soll, erhalten wir außerdem

$$\sum_{j=0}^{n} \delta_{ij} = 1 \qquad \text{für } i = 0, \cdots, n$$

Die Frage ist, ob das Modell

$$\begin{aligned} \min & \sum_{i=0}^{n} \sum_{j=0}^{n} \delta_{ij} c_{ij} \\ \text{u.d.N.} & \sum_{i=0}^{n} \delta_{ij} = 1 \qquad \text{für } j = 0, \cdots, n \\ & \sum_{j=0}^{n} \delta_{ij} = 1 \qquad \text{für } i = 0, \cdots, n \\ & \delta_{ij} \in \{0, 1\} \qquad \text{für } i, j = 0, \cdots, n \end{aligned}$$

unser Problem vollständig beschreibt. Das folgende Beispiel zeigt, dass dies im Allgemeinen nicht so ist.
Wir wählen wieder $n = 5$. Dann wäre eine optimale Lösung $\delta_{03} = \delta_{31} = \delta_{10} = 1$, $\delta_{52} = \delta_{24} = \delta_{45} = 1$ und die übrigen $\delta_{ij} = 0$. In folgender Abbildung sehen wir, dass diese Lösung zwei Teilkreise bildet, damit ist diese Lösung keine zulässige Verteilung der Aufgaben.




Eine zulässige Lösung sollte genau einen Kreis (Zyklus) enthalten. Daher benötigen wir noch Nebenbedingungen, die Teilkreise von optimalen Lösungen nicht zulassen. Solche sind:

$$\sum_{i,j \in S} \delta_{ij} \leq |S| - 1 \qquad \text{für jede echte, nichtleere Teilmenge } S \subsetneq \{0, \cdots, n\}$$

Dabei meint echte Teilmenge, dass $S \neq \{0, \cdots, n\}$. Damit können wir das Maschinenbelegungsplan-Problem wie folgt als ILP formulieren:

**Definition 3.3.3** Das Maschinenbelegungsplan-Problem hat die Form

| | ILP Modell des Maschinenbelegungsplan-Problem |
|---|---|
| min | $\sum_{i=0}^{n} \sum_{j=0}^{n} \delta_{ij} c_{ij}$ |
| u.d.N. | $\sum_{i=0}^{n} \delta_{ij} = 1$ für $j = 0, \cdots, n$ |
| | $\sum_{j=0}^{n} \delta_{ij} = 1$ für $i = 0, \cdots, n$ |
| | $\sum_{i,j \in S} \delta_{ij} \leq \lvert S \rvert - 1$ für jede echte, nichtleere Teilmenge $S \subsetneq \{0, \cdots, n\}$ |
| | $\delta_{ij} \in \{0, 1\}$ für $i, j = 0, \cdots, n$ |

Dies ist außerdem ein Model des Problems des Handelsreisenden, das wie folgt formuliert wird: Gegeben sei eine Anzahl an Städten zusammen mit ihrer jeweiligen Entfernung (z.B. in einer Abstandstabelle). Berechne die kürzeste Route über die Städte so, dass jede Stadt genau einmal besucht wird und Start und Ende identisch sind.

Wir betrachten die folgende Situation. Die entsprechenden Justierungs- und Einrichtungszeiten seien wie folgt gegeben:

| Aufgabe \ Aufgabe | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | - | 1 | 1 | 5 | 4 | 3 | 2 |
| 1 | 1 | - | 2 | 5 | 4 | 3 | 2 |
| 2 | 1 | 5 | - | 4 | 2 | 5 | 4 |
| 3 | 5 | 4 | 6 | - | 6 | 2 | 5 |
| 4 | 5 | 2 | 6 | 3 | - | 5 | 4 |
| 5 | 5 | 3 | 5 | 1 | 5 | - | 3 |
| 6 | 6 | 5 | 4 | 6 | 6 | 5 | - |

Die Anzahl der echten Teilmengen (und damit die Anzahl der entsprechenden Nebenbedingungen) ist $2^{n+1} - 2$, da die Anzahl der Aufgaben $n + 1$ ist (inklusive $P_0$). Die Anzahl der Teilmengen von $\{1, \cdots, n\}$ beträgt $2^{n+1}$ minus die leere Menge und die Menge $\{1, \cdots, n\}$ selbst. In obigem Beispiel ist $n = 6$, daher gibt es $2^7 - 2 = 126$ Nebenbedingungen um die Teilkreise auszuschließen. Zu bemerken bleibt, dass diese Anzahl exponentiell wächst. So ist etwa für $n = 40$, $2^{41} - 2 \approx 2.2 \cdot 10^{12}$.

Wir werden dieses Problem mit dem Branch-and-Bound-Verfahren lösen. Die Tatsache, dass wir dies auf diese Weise tun, heißt aber nicht, dass das Branch-and-Bound-Verfahren die angemessenste Methode ist. Die Anzahl an Iterationen wird enorm groß. Optimale Lösungen können normalerweise nicht in angemessener Zeit gefunden werden. Diese Tatsache hat die bereits erwähnten Heuristiken hervorgebracht. Sie garantieren zwar keine

optimale Lösung, aber liefern in der Regel gute zulässige Lösungen in annehmbarer Zeit. Das Branch-and-Bound-Verfahren kann auch als eine Heuristik verwendet werden. Das Abbruchkriterium könnte etwa abgeschwächt werden (erste zulässige Lösung, nach einer bestimmten Anzahl an Iterationen, o.ä.).

Die allgemeine Idee des Branch-and-Bound-Verfahrens auf das obige Modell angewandt kann wie folgt beschrieben werden: Zunächst lösen wir eine Relaxation des Models. Hier könnte man etwa die Teilkreis Bedingungen weglassen. Wenn die optimale Lösung dieser Relaxation bereits zulässig ist, dann stoppe das Verfahren. Sonst haben wir eine untere Schranke. Weiterhin kann mindestens eine der Variablen, die die Unzulässigkeit verursacht, nicht in der optimalen Lösung vorkommen. Wenn wir ein Teilmodell lösen, bei dem eines dieser Variablen heraus gezwungen wird, dann kann diese Lösung nicht besser sein, als die bisher beste gefundene. Diese Prozedur wird fortgeführt bis die optimale Lösung gefunden wurde.

Wir erläutern dieses Verfahren mit den Zahlen aus obigem Beispiel.

1. Iteration: Das ursprüngliche Modell, ohne die Teilkreis-Nebenbedingungen, hat die Lösung mit zugehöriger unterer Schranke $f = 14$ und den Teilkreisen $P_0 \to P_6 \to P_2 \to P_4 \to P_1 \to P_0$, $P_3 \to P_5 \to P_3$. Im unteren Branch-and-Bound Baum werden diese mit $0-6-2-4-1-0$ und $3-5-3$ notiert (siehe Knoten (1)). Der Teilkreis $3-5-3$ gehört zu $\delta_{35} = 1$ und $\delta_{53} = 1$. Eine optimale Lösung des ursprünglichen Problems kann diesen Teilkreis nicht enthalten. Da wir diese Teiltour entfernen möchten, setzen wir in den nächsten beiden Teilmodellen $\delta_{35} = 0$ und $\delta_{53} = 0$.

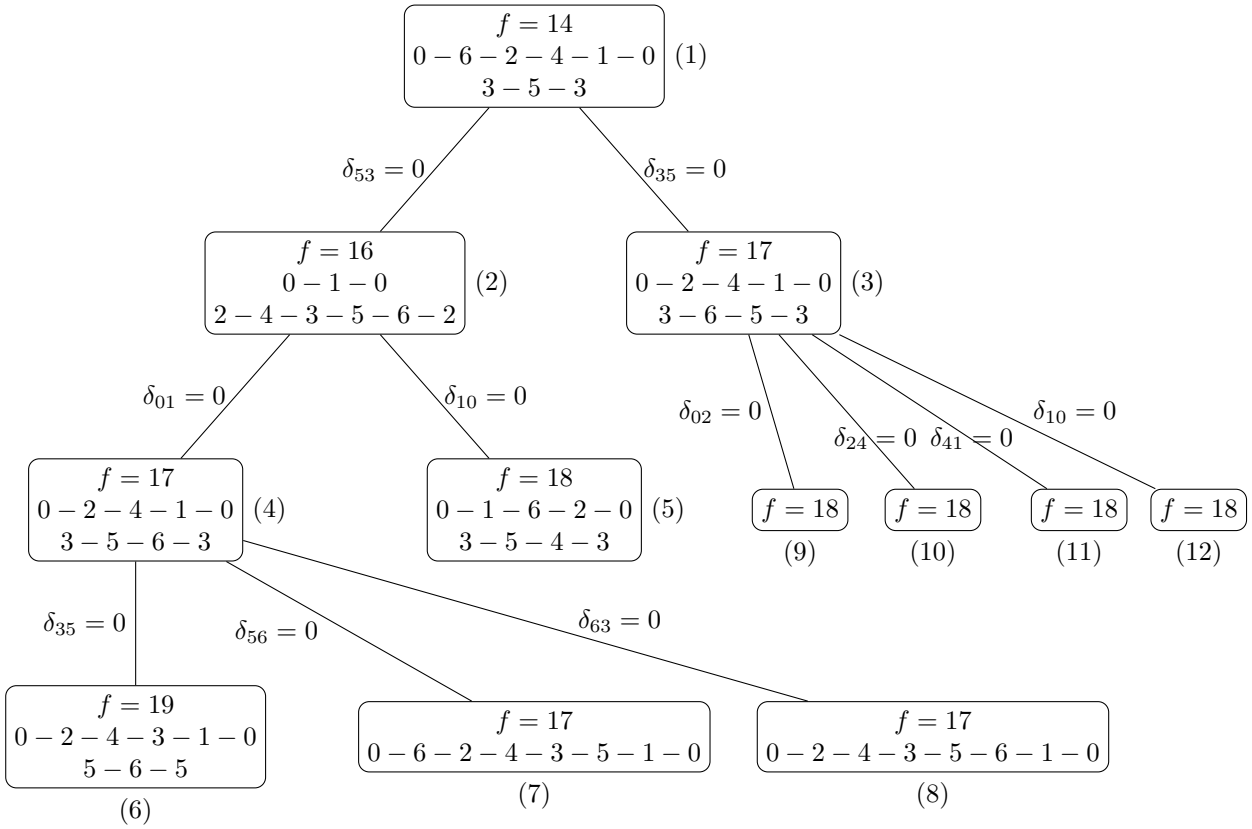
2. Iteration: Die neuen Teilmodelle entsprechen den Knoten (2) und (3). Die optimale Lösung von Modell (2) ist $f = 16$ und hat wieder zwei Teilkreise, $0-1-0$ und $2-4-3-5-6-2$. Modell (3) hat die optimale Lösung $f = 17$ mit den beiden Teilkreisen $0-2-4-1-0$ und $3-6-5-3$. Da Modell (2) eine bessere optimale Lösung als Modell (3) besitzt (besser im Sinne von kleiner), wählen wir (2) um weiter zu verzweigen. Die Verzweigungen entsprechen $\delta_{01}$ und $\delta_{10}$ im Baum.

3. Iteration: Modell (4) hat die Lösung $f = 17$ mit den Teilkreisen $0-2-4-1-0$ und $3-5-6-3$. Modell (5) hat die beiden Teilkreise $0-1-6-2-0$ und $3-5-4-3$ mit $f = 18$. Beachte, dass zu diesem Zeitpunkt des Verfahrens die Teilmodelle (3), (4) und (5) noch nicht ausgelotet sind. Wir wählen (4) um weiter zu verzweigen. Hier wird der Teilkreis $3-5-6-3$ für die weitere Aufspaltung verwendet. Damit erhalten wir drei Verzweigungen.

4. Iteration: Der optimale Wert der Zielfunktion von Modell (6) ist größer, als der von Modell (7) und (8), während Modell (7) und (8) eine zulässige Lösung bieten. Damit ist (6) ausgelotet. Analog gilt, dass Modell (5) ausgelotet ist.

5. Iteration: Der optimale Wert der Zielfunktion ist in den Modellen (9), (10), (11) und (12) jeweils identisch $f = 18$ (hier haben wir im Baum auf die Angabe der Teilkreise verzichtet). Da diese größer sind, als die bereits gefundenen zulässigen und optimalen Werte, sind diese ausgelotet und das Verfahren ist beendet.

Damit erhalten wir den vollständigen Branch-and-Bound Baum:




Damit liefern (7) mit $P_0 \to P_6 \to P_2 \to P_4 \to P_3 \to P_5 \to P_1 \to P_0$ und (8) mit $P_0 \to P_2 \to P_4 \to P_3 \to P_5 \to P_6 \to P_1 \to P_0$ jeweils optimale Lösungen für das Problem. Dabei beträgt die Summe der Einrichtungs- und Justierungszeiten jeweils 17 Minuten.

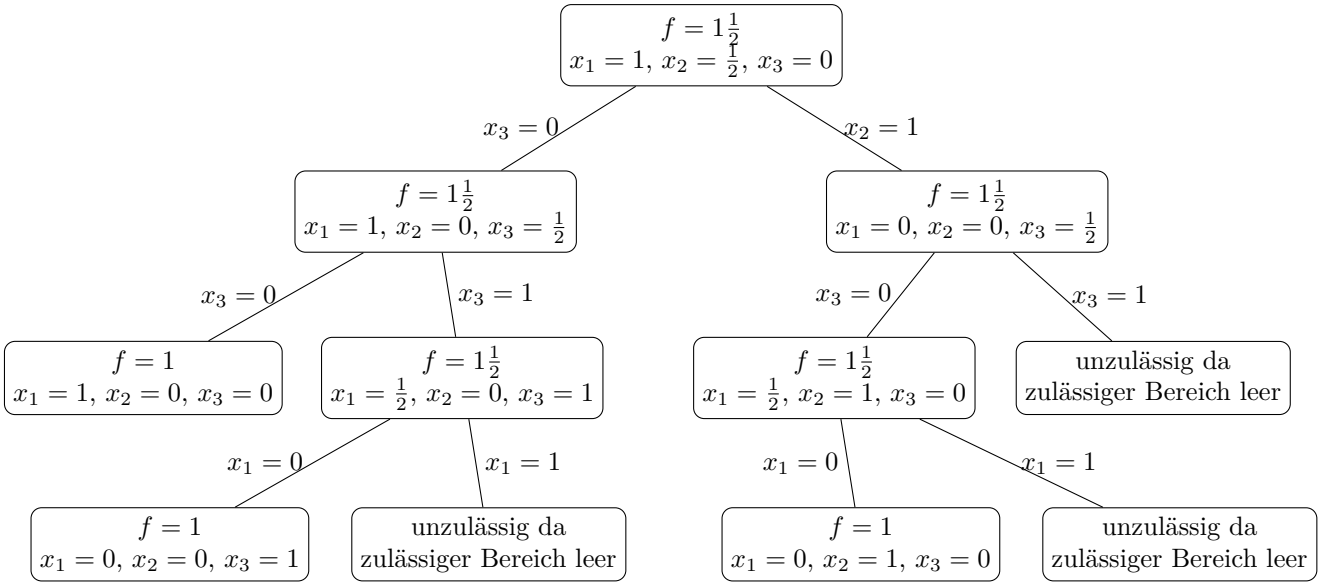
### 3.3.4 Laufzeit des Branch-and-Bound-Verfahren

Zum Abschluss betrachten wir in diesem Abschnitt die Laufzeit des Branch-and-Bound-Verfahrens. Betrachte folgendes Rucksackproblem für $n \geq 3$ ungerade und ganzzahlig:

$$\begin{aligned} &\max x_1 + \cdots + x_n \\ &\text{u.d.N. } 2x_1 + \cdots + 2x_n \leq n \\ &\qquad x_i \in \{0,1\} \quad \text{für } i = 1, \cdots, n \end{aligned}$$

Eine optimale Lösung für dieses Problem kann direkt angegeben werden. Wähle $\lfloor n/2 \rfloor = (n-1)/2$ beliebige $x_i$ und setze diese gleich eins. Die restlichen $\lfloor n/2 \rfloor + 1$ der $x_i$ werden gleich null gesetzt. Der optimale Wert der Zielfunktion ist dann $f^* = \lfloor n/2 \rfloor$. Aber, dieses Modell mit dem Branch-and-Bound-Verfahren zu lösen benötigt eine sehr große Anzahl an Iterationen. Es kann gezeigt werden, dass die Anzahl der zu lösenden Teilprobleme, mindestens $(\sqrt{2})^n$ beträgt (für $n \geq 3$, ungerade und ganzzahlig). [13] Als abschließendes Beispiel sei hier der Branch-and-Bound Baum für n = 3 vorgestellt.

[13] Der Beweis ist in [Sie96] in Kapitel 3 zu finden.




Um eine exakte Lösung für ein gegebenes ILP zu berechnen, kann also das Branch-and-Bound-Verfahren sehr schlecht abschneiden. Aber wenn wir die Abbruchkriterien etwas lockern (z.B. bereits bei der ersten gefundenen ganzzahligen Lösung), haben wir immer noch ein gutes Verfahren. Wie wir außerdem bei den meisten Beispielen gesehen haben, tritt das Worst-Case Szenario (d.h. alle ganzzahligen Punkte werden enumeriert) nicht immer auf.

# 4 Implementierung in Python

In diesem letzten Kapitel wollen wir nun eine Möglichkeit vorstellen, wie wir das Simplex-Verfahren sowie das Branch-and-Bound-Verfahren in Python implementieren können. Python gilt als eine einfach zu erlernende Programmiersprache, die zusammen mit vielen Zusatzkomponenten, sogenannten Modulen, zu einer mächtigen Programmiersprache wird. Da Python nicht kommerziell ist, ist es auch im Sinne der freien Wissenschaft eine gute Alternative zu kommerziellen Programmen. Python wird vielfach angewandt, so basieren etwa Teile von Googles YouTube auf Python. [14]

Wir werden hier Python 3 verwenden. Für schnelle numerische Berechnungen ist das Paket numpy erforderlich. Viele Optimierungsprogramme wurden bereits in Python implementiert. Um unsere Ergebnisse zu vergleichen, werden wir auf das Paket von scipy zurückgreifen. Schließlich vergleichen wir noch unsere Ergebnisse mit Matlab. Grundkenntnisse in Programmierung, wie sie für Matlab an der Universität Konstanz im zweiten und dritten Semester gelehrt werden, werden wir im Folgenden voraussetzen.

In Python werden Schleifen, If-Bedingungen und Funktionen stets mit dem jeweiligen Befehl und einem Doppelpunkt begonnen und im Anschluss wird eingerückt gearbeitet. Dadurch muss kein manuelles Ende der Schleife angegeben werden. Als Beispiel betrachten wir eine If-Bedingung.

```
if Bedingung 1:
        do
elif Bedingung 2:
        do
else:
        do
```

Ein weiterer Hauptunterschied ist, dass wir für nummerische Berechnungen numpy.arrays verwenden die das Matlab-Äquivalent zu Matrizen bilden. Hier gibt es 1-D-Arrays, 2-D-Arrays, usw. Dabei ist ein Vorteil bei der Benutzung von 1-D-Arrays (für Vektoren), dass diese zwar stets „liegend“ angezeigt werden, bei der entsprechenden Matrixmultiplikation aber immer (sofern die Dimension stimmt) so transponiert werden wie die mathematische Berechnung definiert ist. Als Beispiel betrachten wir

```
import numpy as np

A = np.array([[1,2,3],[4,5,6],[7,8,9]])
x_1 = np.array([[1],[2],[3]])
x_2 = np.array([1,2,3])

b_1 = np.dot(A,x_1)
```

[14] Vergleiche Einleitung von [Wei16] und [Pyth]

```
b_2 = np.dot(A,x_2)
b_3 = np.dot(x_1,A)
b_4 = np.dot(x_2,A)
```

Dabei erhalten wir die zu erwartenden Ergebnisse der Matrixmultiplikation: für b_1 (einen „stehenden“ Vektor), für b_2 und b_4 (jeweils „liegende“ Vektoren) das richtige Ergebnis, für b_3 eine Fehler Meldung, weil hier die Dimensionen nicht stimmen.
Eine ausführliche Dokumentation der Pakete Numpy und Scipy ist unter [Num] und [Sci] zu finden.

## 4.1 Simplex

Zuerst importieren wir die benötigten Pakete. Hierbei (wie auch im Folgenden) werden wir numpy stets als np importieren. Dies soll verhindern, dass die Eindeutigkeit von Befehlen verloren geht und sollte vor allem immer dann gemacht werden, wenn größere Pakete importiert werden. Weiter importieren wir auch das time-Paket um die benötigte Zeit zu messen.

```
import numpy as np
import time
```

Als nächstes wollen wir die Simplex Funktion definieren. Funktionen, die in Python definiert werden, werden stets mit

```
def Name(...):

        return{output1, output2,...}
```

eingeleitet, wobei der Doppelpunkt nicht vergessen werden darf. Dabei steht in den Klammern das Input. Danach wird eingerückt gearbeitet. Die definierte Funktion endet für Python dort, wo nicht weiter eingerückt gearbeitet wurde. Der Output wird in return festgehalten. Um diesen aufzurufen schreiben wir

```
Name['output1']
```

Wir beginnen also mit:

```
def simplex(c, N, b, disp=0, maxiter = 10000):
```

Dabei sind c, N und b jeweils np.arrays die das LP repräsentieren. Die beiden folgenden Argumente disp=0 und maxiter=10000 sind optionale Argumente. Wenn beim Aufruf der Funktion nur die drei Input-Argumente angegeben werden, wählt der Algorithmus automatisch die oben angegebenen. Das disp soll die beiden Argumente 0 und 1 haben. Dabei soll bei Null kein Output angezeigt werden. Dies realisieren wir später im Algorithmus. Das maxiter soll als eine Grenze für die Anzahl an Iterationen dienen, die wir nicht überschreiten wollen. Als nächstes kommt dann der Hilfetext, der dem User angezeigt wird, wenn er

```
help(Funktion)
```

eingibt. Dieser Text wird wie folgt definiert:

```
def Name(...):
        """
        Hier soll ein Text stehen
        """
```

Der help (Funktion)-Befehl liefert dann den gesamten Text, der zwischen den drei Anführungszeichen steht. Der vollständige Hilfetext für das Simplexverfahren ist im Anhang, dort ist auch der vollständige Code zu finden. Der Hilfetext sollte stets Input und Output, eine kurze Erklärung wofür die Funktion gedacht ist und welche Pakete bzw. welche anderen Funktionen benötigt werden, enthalten.

Als nächstes definieren wir unsere initial Variablen:

```
#set initial values
start = time.time()
C = N.copy()                    #define nonbasicmatrix C
[n,m] = np.shape(C)             #check the dimension of N
B = np.eye(n)                   #define basicmatrix B
A = np.hstack([C,B])            #define matrix A
x_b = b                         #define basic x
z_n = -c
#define basic and nonbasic
CB = np.arange(m+1,m+n+1)       #basic
CN = np.arange(1,m+1)           #nonbasic

count = 0                       #define counter
x_opt = np.zeros(m)             #defin initial x_opt
bound = 0                       #define initial boundness
feasible = 0                    #define initial feasible
```

Dabei kommentieren wir, indem wir vor dem gewünschten Kommentar ein # schreiben (dies entspricht in Matlab %). Als erstes definieren wir uns einen Startwert wie angegeben. Von unserem Input N machen wir eine Kopie, anstatt eine neue Variable zu definieren. Dies hat den Hintergrund, dass Python in der Funktion sonst das N genauso ändern würde, was wir nicht wollen. Der numpy-Befehl np.shape ist, um die Größe zu bestimmen, die unser Input liefert, np.eye liefert eine Einheitsmatrix der Dimension n (also ein 2-D-Array mit Einsen auf der Diagonalen und Nullen sonst) und np.hstack definiert uns ein neues 2-D-Array. Der Befehl np.arange(1,m) liefert ein 1-D-Array der von 1 bis m-1. Damit definieren unser CB und CN unsere Indice-Mengen. Schließlich liefert der Befehl np.zeros(m) eine 1-D-Array aus Nullen der Länge m. Zu beachten ist noch, dass Python seine Indizierung stets bei Null beginnt. Wir definieren noch bound und feasible, die uns im Laufe das Algorithmus mitteilen sollen, welchen Status das LP hat. Dabei soll bound=0 für ein normales LP stehen, dessen Lösung gefunden wurde; bound=1 für (primal) unbeschränkte Probleme stehen; bound=2 für dual unbeschränkte (und damit ist dann der zulässige Bereich leer) und bound=3 anzeigen, dass die maximale Anzahl an Iterationen erreicht wurde. Für feasible wählen wir feasible=1 für primal zulässige, feasible=2 für dual zulässige und feasible=3 weder noch, geartete Probleme. Als nächstes prüfen wir die Zulässigkeit.

```
#check feasible
if np.where(x_b < 10**(-10), True, False).any() == False:
    feasible = 1
elif np.where(z_n < 10**(-10), True, False).any() == False:
    feasible = 2
else:
    feasible = 3
```

Da wir hier einen Vektor komponentenweise überprüfen wollen, nutzen wir den Befehl np.where. Dabei erhalten wir einen np.array derselben Größe, der für jeden Eintrag, entsprechend der Bedingung, ein True bzw. False setzt. Als Beispiel liefert

```
x_b = np.array([-1, 1, 0])
test = np.where(x_b < 10**(-10), True, False)
```

den Output

```
print(test)
array([ True, False, False], dtype=bool)
```

Mit .any()==False prüfen wir ob mindestens ein Eintrag die gesuchte Bedingung erfüllt. Aus numerischen Gründen wählen wir nicht Null als Schranke, sondern Zahlen, die nahe bei Null sind (hier $10^{-10}$; in Python werden Exponenten mit ** geschrieben). Als nächstes lösen wir das Problem wie im Algorithmus in 2.3.4 angegeben.

```
#solve with simplex
if feasible == 1:
    solve = solve_primal(C, n, m, B, A, x_b, z_n, CB, CN, \
                         count, x_opt, bound, maxiter)

elif feasible == 2:
    solve = solve_dual(C, n, m, B, A, x_b, z_n, CB, CN, \
                         count, x_opt, bound, maxiter)

elif feasible == 3:
    #substitute
    x_b_sub = np.ones(len(b))
    solve_sub = solve_primal(C, n, m, B, A, x_b_sub, z_n, CB, \
                         CN, count, x_opt, bound, maxiter)

    count = solve_sub['iterations']

    if solve_sub['bound'] == 1:
        bound = 1

    elif solve_sub['bound'] == 0:
        CN = solve_sub['CN']
        CB = solve_sub['CB']
        index_CB = CB - 1
        index_CN = CN - 1
        B = np.take(A, index_CB, axis = 1)
```

```
        C = np.take(A, index_CN, axis = 1)
        z_n = solve_sub['z_n']
        x_b = np.linalg.solve(B, b)
        solve = solve_dual(C, n, m, B, A, x_b, z_n, CB, CN, \
                                    count, x_opt, bound, maxiter)
```

Die beiden Solver (der eigentliche Simplex-Algorithmus) stellen wir unten vor. Sei es, dass das LP weder primal noch dual zulässig ist, führen wir einen 2-Phasen-Algorithmus durch wie in Abschnitt 2.3.5 beschrieben. Der Befehl np.take definiert uns die neuen Basis und Nichtbasismatrizen, indem wir von A die Spalten (axis=1) die in index_CB aufgeführt sind wählen. Zum Schluss definieren wir noch die Ergebnisse und geben den Output an.

```
if feasible != 3:
    bound = solve['bound']
    x_opt = solve['x_opt']
    count = solve['iterations']
    CN = solve['CN']
    CB = solve['CB']
    x_b = solve['x_b']
    z_n = solve['z_n']
    f_opt = np.dot(c.T,x_opt)
elif feasible == 3 and bound == 0:
    bound = solve['bound']
    x_opt = solve['x_opt']
    count = solve['iterations']
    CN = solve['CN']
    CB = solve['CB']
    x_b = solve['x_b']
    z_n = solve['z_n']
    f_opt = np.dot(c.T,x_opt)

comptime = (time.time() - start)

#output after checking bound
if bound == 0:
    return{'feasible': feasible, 'bound': bound, 'x_opt': \
       x_opt, 'f_opt': f_opt, 'iterations': count, 'time': \
       comptime, 'A': A, 'CN': CN, 'CB': CB, 'x_b': x_b, 'z_n':\
       z_n, 'c': c, 'b': b}

elif bound == 1:
    x_opt = 0
    return {'iterations': count, 'f_opt': "no solution found",\
     'time': comptime, 'bound': bound, 'x_opt': x_opt}

elif bound == 2:
    x_opt = 0
    return {'iterations': count, 'f_opt': "no solution found",\
```

```
        'time': comptime, 'bound': bound, 'x_opt': x_opt}

    elif bound == 3:
        x_opt = 0
        return {'iterations': count, 'f_opt': "no solution found",\
         'time': comptime, 'bound': bound, 'x_opt': x_opt}
```

Als nächstes implementieren wir den eigentlichen Simplex-Algorithmus. Bevor wir dies tun können, müssen wir noch eine eigene Simplex-Division definieren, um der Konvention $\frac{0}{0} = 0$ (vgl 2.3.2, 4. Schritt) gerecht zu werden. Eine erste Variante wäre

```
def simplexdivision(a,b):
    if a==0 and b==0:
        c = 0
    elif a < 0 and b==0:
        c = -np.inf
    elif a > 0 and b==0:
        c = np.inf
    else:
        c = a/b
    return c
```

Hier müssten wir aber für unsere np.arrays eine for-Schleife einbauen. Da Schleifen in Python aber allgemein teuer sind wählen wir eine Variante, die die Division in einem gesamte np.array durchführt.

```
def division( a, b ):
    """
    ignore / 0,
    division( np.array([-1, 0, 1]), np.array([0,0,0]) )
        -> [-inf, 0, inf]
    """
    with np.errstate(divide='ignore', invalid='ignore'):
        a[ abs(a) <= 10**(-10) ] = 0
        b[ abs(b) <= 10**(-10) ] = 0
        c = np.true_divide( a, b )
        c[  np.isnan( c )] = 0  #NaN
    return c
```

Dabei liefert np.errstate die Möglichkeit, mit bekannten Fehlern umzugehen bzw. diese umzudefinieren. Dabei runden wir beim Input Zahlen die sehr nahe bei Null sind. Das Output soll uns das übliche Ergebnis bei der Division liefern, außer Zähler und Nenner sind gleich Null (dies liefert in Python NaN). In diesem Fall wollen wir die Null haben, das erreichen wir mit der letzten Zeile. Da es nur ein Output gibt, müssen keine geschweiften Klammern (also { und }) geschrieben werden.

Damit können wir nun den primalen Simplex-Algorithmus implementieren.

```
def solve_primal(C, n, m, B, A, x_b, z_n, CB, CN, count, x_opt,\
        bound, maxiter):
    """
```

```
    Hilfetext
    """
```

Die While-Schleife können wir auch wieder auf zwei Wegen definieren, entweder

```
while np.where(z_n < 10**(-10), True, False).any() == True \
        and bound == 0:
    #select entering variable: (choose j like in Bland)
    j_search = np.argmin(z_n)
    j = CN[j_search]
```

oder

```
ma_test = np.ma.masked_where(z_n >= -10**(-10), z_n, copy=False)
while ma_test.count() > 0 and bound == 0 and count <= maxiter:
    #select entering variable: (choose j like in Bland)
    j_search = np.ma.where(ma_test == ma_test.min())[0][0]
    j = CN[j_search]
```

Dabei wird im ersten Fall beim Test auf die Zulässigkeit geprüft, unten wird ein neues mask.array erstellt. Dabei werden nur dort Werte gespeichert, wo die Bedingung noch verletzt ist. Da es insbesondere bei großen Problemen nützlich ist, wenn weniger Variablen abgesucht werden müssen, wählen wir die zweite Variante. Nachdem wir die Eintrittvariable gewählt haben, müssen wir die primale Schrittrichtung $\Delta x_{\mathcal{B}}$ berechnen. Dafür benötigen wir zunächst den k-ten Einheitsvektor.

```
#compute kth unit vector, where k is the position of j in CN
e_k = np.eye(n)
k = j_search
e_k[0] = 0
e_k[1] = 1

#possible improvement
k = j_search
e_k = np.eye(m,1,1-(k+1))
e_k.shape = (m)
```

Auch hier präsentieren wir wieder zwei Möglichkeiten. Die erste ist die wohl Intuitivere. Aber np.eye bietet eben auch mit den optionalen Parametern die Möglichkeit, den k-ten Einheitsvektor direkt anzugeben. Wir müssen im Anschluss nur noch aus dem 2-D-Array (eine $m \times 1$-Matrix) ein 1-D-Array der Länge $m$ machen, was wir mit dem np.shape(m) Befehl erreichen. Damit können wir nun $\Delta x_{\mathcal{B}}$ berechnen.

```
#compute primal step direction deltax_b (deltax_b = B^(-1)Ne_k)
C_tilde = np.dot(C,e_k)
deltax_b = np.linalg.solve(B, C_tilde)
```

Hier ist np.linalg.solve der Solver für LGS von Numpy. Als nächstes berechnen wir die primale Schrittlänge, und die Austrittsvariable und testen, ob das LP unbeschränkt ist. Auch hier präsentieren wir zwei Varianten.

```
#comoute primal step lenght and choose leaving variable
T = np.zeros((n,1))
```

```
for i in range(0,n):
    T[i] = simplexdivision(deltax_b[i],x_b[i])

if np.where(T <= 10**(-10), True, False).all():
    count = count -1
    bound =  1

for i in range(0,n):
    if T[i] == max(T):
        a = i
i = CB[a]
t = 1 /(max(T))



#possible improvement
T = division(deltax_b,x_b)

if np.where(T <= 10**(-10), True, False).all():
    count = count -1
    bound =  1

T_bar = np.argwhere(T == np.amax(T))
Max_T_bar = np.min(T_bar)
i = CB[Max_T_bar]
t = 1/(np.max(T))
```

Dabei beruht die erste Variante auf der ersten Variante der Simplexdivision, wie wir sie oben definiert haben. Wir wählen hier die zweite, da sie die schnellere Variante der Simplexdivision nutzt und eine for-Schleife vermieden werden kann (siehe auch Abschnitt 4.3.1.
Der nächste Schritt im Algorithmus ist die Berechnung der dualen Schrittrichtung und der dualen Schrittlänge.

```
#compute lth unit vector, where l is the position of i in CB
l = Max_T_bar
e_l = np.eye(n,1,1-(l+1))
e_l.shape = (n)

#compute dual step direction deltaz_n (deltaz_n=-(B^(-1)N)^Te_l)
v = np.linalg.solve(B.T,e_l)
deltaz_n = np.dot(-C.T,v)

#compute dual step length
s = z_n[np.argmin(z_n)]/deltaz_n[np.argmin(z_n)]
```

Zum Schluss noch das Update der Variablen.

```
#update
x_b = (x_b - t * deltax_b) + t * e_l
```

```
z_n = (z_n - s * deltaz_n) + s * e_k
CB[l] = j
CN[k] = i


for i in range(0,n):
    B[:,i] = A[:,CB[i]-1]
for i in range(0,m):
    C[:,i] = A[:,CN[i]-1]

#possible improvement
index_CB = CB - 1
index_CN = CN - 1
B = A[:, index_CB]
C = A[:, index_CN]

#possible improvement
B[:,l] = A[:, j-1]
C[:,k] = A[:, i-1]

count += 1
ma_test = np.ma.masked_where(z_n >= -10**(-10), z_n, copy=False)
```

Hier haben wir gleich drei Varianten für das Uptdate der Basis- und Nichtbasismatrizen. Dabei entscheiden wir uns für die dritte Variante, da sie erstens am schnellsten ist und zweitens am wenigsten Code-Zeilen benötigt. Der Befehl count += 1 ist typisch für Python und wird für count = count + 1 verwendet.
Abschließend definieren wir noch unser Output und beenden damit die Implementierung des primalen Simplex.

```
if count == maxiter:
    bound = 3
for i in range(0,m):
    if i+1 in CB:
        x_opt[i] = x_b[np.where( CB == i+1 )]
    else:
        x_opt[i] = 0

return {'bound': bound, 'x_opt': x_opt, 'iterations': count, \
        'CN': CN, 'CB': CB, 'x_b': x_b, 'z_n': z_n}
```

Den dualen Simplex-Algorithmus können wir analog implementieren. Dieser ist ebenfalls im Anhang zu finden. Für das Branch-and-Bound-Verfahren benötigen wir noch eine kleine Abwandlung des Simplex-Algorithmuses, da wir hier nachträglich Nebenbedingungen hinzufügen (vergleiche 2.4.3 den 5. Punkt). Auch dieser ist im Anhang zu finden.

## 4.2 Branch-and-Bound

Als erstes laden wir wieder unsere Pakete und definieren im Anschluss eine Funktion um zu bestimmen, ob ein gefundenes Optimum ganzzahlig ist.

```
import time
import numpy as np
from simplex import simplex
from simplex import dsimplex
```

Hier ist zu beachten, dass die Datei simplex.py im selben Ordner gespeichert sein muss.

```
def isinteger(x):
    """
    verifies elementwise to an integer
    args:       an array
    returns:    a mask array
    """
    x_1 = np.around(x, decimals = 6)
    y = np.mod(x_1,1)
    ma = np.ma.masked_where(abs(y) <= 10**(-6), y, copy=False)
    return ma
```

Dabei rundet der Befehl np.round das entsprechende np.array auf sechs Nachkommastellen und np.mod führt auf ein np.array eine Ganzzahldivision durch. Da eine Zahl $x_i$ ganzzahlig ist, wenn der Rest bei Division durch 1 gleich Null ergibt, führen wir die Division wie oben angegeben durch. Die Schranke von $10^{(-6)}$ wirkt nicht besonders scharf, ist aber auch die genaueste Einstellung, die man bei Matlab vornehmen kann. Da wir für die numerischen Tests verschiedene Varianten des Branchings testen wollen, definieren wir noch eine Funktion, die das strong branching repräsentiert.

```
def strong_branching(P, ma, c):
    index0 = ma.nonzero()
    index1 = index0[0].astype(int)
    len_index = len(index1)
    len_c = len(c)
    Z = np.vstack([index1, np.zeros(len_index), \
        np.zeros(len_index)])
```

Dabei ist der Befehl np.vstack ähnlich wie der oben eingeführte np.hstack, mit dem Unterschied, dass hier das neue Array untereinander zusammengefügt wird, statt nebeneinander. Als nächstes gehen wir in eine for-Schleife und berechnen mit dem dualen Simplex die optimalen Werte der Zielfunktionen. Hier sollen uns 10 Iterationen genügen. Falls dabei eine ganzzahlige Lösung gefunden wird, soll diese ausgegeben werden. Die Ausgabe ist dann der Index, der gewählt werden soll, um ein Problem weiter zu verzweigen (branchen).

```
for i in range(0,len_index):
    l = Z[0][i]
    l = int(l)
    input_v_1 = np.eye(len_c,1,1-(l+1))
    input_v_1.shape = (len_c)
```

```
        input_b_1 = np.floor(P['x_opt'][i])

        input_v_2 = -1*input_v_1
        input_b_2 = -1*(input_b_1 + 1)

        Node_1 = dsimplex(P, input_v_1, input_b_1, maxiter = 10)
        bound_1 = Node_1['bound']
        f_opt_1 = Node_1['f_opt']

        if bound_1 == 2:
            Z[1][i] = np.nan
        elif isinteger(Node_1['x_opt']).count() == 0:
            z = int(Z[0][i])
            return z
            break
        else:
            Z[1][i] = f_opt_1

        Node_2 = dsimplex(P, input_v_2, input_b_2, maxiter = 10)
        bound_2 = Node_2['bound']
        f_opt_2 = Node_2['f_opt']

        if bound_2 == 2:
            Z[2][i] = np.nan
        elif isinteger(Node_2['x_opt']).count() == 0:
            z = int(Z[0][i])
            return z
            break
        else:
            Z[2][i] = f_opt_2

    if np.isnan(Z[1]).all():
        z = int(Z[0][0])
    elif np.isnan(Z[2]).all():
        z = 0
    else:
        z_1 = np.nanargmin(Z[1])
        z_2 = np.nanargmin(Z[2])
        if Z[1][z_1] >= Z[1][z_2]:
            z = int(Z[0][z_1])
        else:
            z = int(Z[0][z_2])
    return z
```

Als nächstes implementieren wir das Branch-and-Bound-Verfahren wie in Algorithmus 3.2.1. Dieses soll ein Initial-Verfahren beinhalten, das eine erste zulässige Lösung findet und möglicherweise dadurch bereits eine gute obere Schranke. Dieses Initial-Verfahren ist im Anhang zu finden, ist aber identisch zum eigentlichen Algorithmus. Es stoppt aber bereits nach der ersten gefundenen zulässigen (also ganzzahligen) Lösung.

Wir starten wie üblich

```
def b_and_b(c, N, b, disp=0, variable_search = 0, \
        initial = 'deepfirst', nodesearch = 'bestsolution',\
        maxiter = 10**(4)):
```

Die Eingabe c, N und b repräsentieren wieder unser LP. Die beiden optionalen Argumente disp=0 und maxiter sind genau wie oben beim Simplex-Verfahren. Für variable_search wollen wir fünf verschiedene Möglichkeiten angeben (diese geben vor, wie verzweigt werden soll). Mit initial wollen wir entweder mit einer deepfirst (LIFO) Methode starten oder nach der aktuell besten Lösung (MUB) ein initial Verfahren starten. Das nodesearch gibt vor, welche Knoten als nächstes untersucht werden sollen, auch hier sollen die beiden Möglichkeiten deepfirst (LIFO) und aktuell beste Lösung (MUB) möglich sein. Als nächstes definieren wir unsere initial Werte und starten mit einem Initial-Verfahren.

```
start = time.time()
count = 0
f_opt = np.array([-1* np.inf])
x_opt = 0
index =  set([0])
finish = set()
f_o = np.array([-1* np.inf])
bound = 0

if initial == 'deepfirst':
    P_initial = initial_b_and_b(c, N, b, disp = 0, \
        variable_search = 0, initial = 'deepfirst')
elif initial == 'bestsolution':
    P_initial = initial_b_and_b(c, N, b, disp = 0, \
        variable_search = 0, initial = 'bestsolution')
```

Der Befehl set() definiert uns die leere Menge, set([0]) die Menge die die Null enthält. Als nächstes prüfen wir, was das Initial-Verfahren für Ergebnisse liefert. Dabei prüfen wir zunächst auf Beschränktheit, da wir das Verfahren beenden können, wenn bereits das Initial-Verfahren gezeigt hat, dass es keine Lösung gibt.

```
if P_initial['bound'] == 1:
    comptime = (time.time() - start)
    bound = 1
    return {'time': comptime, 'iterations': \
    P_initial['iterations'], 'f_opt': f_opt, 'bound': bound}

if P_initial['bound'] == 2:
    comptime = (time.time() - start)
    bound = 2
    return {'time': comptime, 'iterations': \
    P_initial['iterations'], 'f_opt': f_opt, 'bound': bound}

if P_initial['bound'] == 4:
    comptime = (time.time() - start)
```

```
    bound = 4
    return {'time': comptime, 'iterations': \
    P_initial['iterations'], 'f_opt': f_opt, 'x_opt': x_opt, \
    'bound': bound}
```

Dabei hat bound hier dieselbe Bedeutung wie beim Simplex. Neu ist nur bound = 4: dies tritt ein, wenn das LP keine ganzzahlige Lösung hat. In all diesen drei Fällen können wir den Algorithmus beenden. Als nächstes aktualisieren wir unsere Werte und lösen das erste LP.

```
count_initial = P_initial['iterations']
f_opt = P_initial['f_opt']
x_opt = P_initial['x_opt']
Node = simplex(c, N, b)

#define position
pos = np.array([[0],[0],[0],[Node],[Node['f_opt']]])
```

Die Variable pos soll unseren Baum repräsentieren. Dabei steht die erste Zeile für das Problem, die zweite Zeile für die Ebene und die dritte Zeile dafür, ob wir links oder rechts stehen. Die vierte Zeile speichert unser gelöstes LP, die fünfte speichert den entsprechenden Wert der Zielfunktion. Als nächstes können wir in die while-Schleife gehen, einen Knoten wählen und die lokale Schranke definieren.

```
while index != finish:
    #select k
    #bestsolution choose the maximum local upper bound,
    #deepfirst choose the most deep and left node first
    if nodesearch == 'bestsolution':
        axis = pos[4].astype(float)
        if np.nanmax(axis) < f_opt:
            break
        level_search = np.nanargmax(axis)
    elif nodesearch == 'deepfirst':
        level_search = np.argmax(pos[1])
    k = int(pos[0][level_search])

    #bounding
    f_o = pos[4][k]
    integer_test = isinteger(pos[3][k]['x_opt'])
    count = int(np.max(pos[0]))
    if count == maxiter:
        bound = 3
        break
```

Da wir später die Werte der Zielfunktion als np.nan speichern, müssen wir hier aufpassen. Python unterscheidet wesentlich stärker zwischen Typen von Zahlen als Matlab. Sobald in der fünften Zeile der Wert np.nan auftritt, wird die gesamte erste Zeile als float gespeichert, also als Gleitkommazahl. Dies würde einen Fehler liefern, wenn wir diese weiter unten beim bounding als Index wählen. Deswegen müssen wir, nachdem wir

den Index gewählt haben, k wieder zu einer ganzen Zahl umwandeln, was der Befehl int erledigt. Zusätzlich beenden wir die while Schleife, falls unsere aktuell beste Lösung der LP-Relaxationen schlechter ist als unsere aktuell beste globale Schranke, da wir keine bessere Lösung mehr finden können (vergleiche Bemerkung 3.1.1). Schließlich liefert der Befehl np.nanargmax den Index, indem das Maximum angenommen wird, und ignoriert dabei die nan-Werte.

Als nächstes kommt die Auslotung.

```
#fathomed
if f_o <= f_opt:
    pos[1][k] = 0
    pos[3][k] = 0
    pos[4][k] = np.nan
    index.discard(k)
elif f_o > f_opt and integer_test.count() == 0:
    f_opt = np.around(f_o, decimals = 6)
    x_opt = np.around(pos[3][k]['x_opt'], decimals=12)
    pos[1][k] = 0
    pos[3][k] = 0
    pos[4][k] = np.nan
    index.discard(k)
elif pos[3][k]['bound'] == 2:
    pos[1][k] = 0
    pos[3][k] = 0
    pos[4][k] = np.nan
    index.discard(k)
```

In allen drei Auslotungsfällen, löschen wir k aus dem Index mit dem Befehl index.discard(k). Weiter (vor allem um Arbeitsspeicher zu sparen) löschen wir die entsprechenden Einträge bzw. setzen den Wert np.nan für den Wert der Zielfunktion ein. Im zweiten Auslotungsfall runden wir unsere gefundene Lösung entsprechend der gewünschten Genauigkeit von $10^{(-6)}$ mit dem Befehl np.around. Dabei ist das erste Argument das entsprechende Array, das zweite optionale Argument legt fest, auf wieviele Nachkommastellen gerundet werden soll.

Falls keiner der Auslotungsfälle eintritt, kommt nun das Branching. Da wir hier mehrere Möglichkeiten haben, wobei wir uns über das optionale Argument variable_search vorher festgelegt haben, lösen wir dies über eine if-Bedingung

```
#branching
#choose an index
if variable_search == 0:
    #choose the index where the variable is furthest
    #away from integer
    i_search = np.ma.argmin(abs(integer_test -0.5))
elif variable_search == 1:
    #choose the first minimal index
    i_search = np.ma.argmin(integer_test)
elif variable_search == 2:
```

```
    #choose the first maximal index
    i_search = np.ma.argmax(integer_test)
elif variable_search == 3:
    #choose the first possible index
    i_search = min(np.ma.flatnotmasked_edges(integer_test))
elif variable_search == 4:
    #choose with strong branching
    if integer_test.count() == 1:
        i_search = min(np.ma.flatnotmasked_edges(integer_test))
    else:
        i_search = strong_branching(pos[3][k], integer_test, c)
```

Der Befehl np.ma.flatnotmasked_edges liefert hierbei den ersten und letzten Index. Da wir den ersten zulässigen Index suchen, genügt es, hier das Minimum zu wählen. Als nächstes definieren wir unsere neuen Teilprobleme und lösen diese gleich. Hier exemplarisch für den Fall $x_i \leq \lfloor x_i^k \rfloor$. Der andere Fall geht analog.

```
max_index = max(pos[0])

l = i_search
input_v_1 = np.eye(len(c),1,1-(l+1))
input_v_1.shape = (len(c))
input_b_1 = np.floor(pos[3][k]['x_opt'][i_search])
Node_1 = dsimplex(pos[3][k], input_v_1, input_b_1, maxiter=1000)
if Node_1['bound'] == 0:
    if Node_1['f_opt'] <= f_opt:
        add_1 = np.array([[(max_index)+1],[0],[0],[0],[np.nan]])
    else:
        add_1 = np.array([[(max_index)+1],[pos[1][k]+1],[0], \
        [Node_1], [Node_1['f_opt']]])
        index.add((max_index)+1)
#if dsimplex maxiter reached define new variable search
elif Node_1['bound'] == 3:
    variable_search = np.mod(variable_search+1, 4)
else:
    add_1 = np.array([[(max_index)+1], [0], [0], [0], [np.nan]])
```

Der Befehl np.floor entspricht hier der unteren Gaußklammer. Im Fall, dass der duale Simplex zu viele Iterationen benötigt, ändern wir die Auswahl beim Branching, in der Hoffnung, einen besseren Zweig im Baum zu finden. Im Anschluss an das Branching führen wir noch das Update durch.

```
pos = np.hstack([pos,add_1,add_2])
index.discard(k)
pos[1][k] = 0
pos[3][k] = 0
pos[4][k] = np.nan
```

Ist die while-Schleife beendet, definieren wir noch unser Output und beenden damit den Algorithmus, der in seiner Gesamtheit im Anhang zu finden ist.

```
count = int(np.max(pos[0]))
comptime = (time.time() - start) + P_initial['time']

return {'time': comptime, 'iterations': count, 'f_opt': f_opt, \
        'x_opt': x_opt, 'bound': bound}
```

## 4.3 Numerische Tests

### 4.3.1 Simplex

Wir starten zunächst mit einem kleinen Vergleich. Dafür testen wir, was es für einen Unterschied macht, wenn wir statt

```
#comoute primal step lenght and choose leaving variable
T = np.zeros((n,1))
for i in range(0,n):
    T[i] = simplexdivision(deltax_b[i],x_b[i])

if np.where(T <= 10**(-10), True, False).all():
    count = count -1
    bound =  1

for i in range(0,n):
    if T[i] == max(T):
        a = i
i = CB[a]
t = 1 /(max(T))
```

eine Variante ohne for-Schleife wählen

```
#comoute primal step lenght and choose leaving variable
T = division(deltax_b,x_b)

if np.where(T <= 10**(-10), True, False).all():
    count = count -1
    bound =  1

T_bar = np.argwhere(T == np.amax(T))
Max_T_bar = np.min(T_bar)
i = CB[Max_T_bar]
t = 1/(np.max(T))
```

Dazu wählen wir Zufallsmatrizen aus, die ein entsprechendes LP

$$\begin{aligned} &\max c^T x \\ &\text{u.d.N. } Nx \leq b \\ &\qquad x \geq 0 \end{aligned}$$

mit $c \in \mathbb{R}^m$, $N \in \mathbb{R}^{m \times n}$ und $b \in \mathbb{R}^n$, repräsentieren, und lösen dieses. Dies wiederholen wir 100 mal, um etwas mehr Aussagekraft über die Ergebnisse zu erhalten. Um Zufallszahlen zu erzeugen, verwenden wir das Paket random und von diesem das Modul seed() um zu verhindern, dass immer die gleichen Zahlen auftauchen (vergleich [The14], Kapitel 3). Ein entsprechender Code könnte folgendermaßen aussehen.

```
import numpy as np
import random
random.seed()

c = np.zeros(m)
for i in range(0,m):
    c[i] = random.randint(-integer,integer)



b = np.zeros(n)
for i in range(1,n):
    b[i] = random.randint(-integer,integer)


N = np.zeros((n,m))
for i in range(0,n):
    for j in range(0,m):
        N[i][j] = random.randint(-integer,integer)
```

Dabei liefert random.randint(-integer,integer) eine ganze zufällige Zahl zwischen -integer und integer, die zusammen mit $m$ und $n$ entsprechend zu wählen ist. Wir erhalten für $m = 100$, $n = 120$, und integer $= 10$ die folgenden Mittelwerte:

| | Zeit | Iterationen | Wert der Zielfkt |
|---|---|---|---|
| mit for-Schleife | 0.231 | 277.29 | 307.287 |
| ohne for-Schleife | 3.927 | 277.29 | 307.287 |

Ein eindrucksvolles Ergebnis, durch welches wir sofort erkennen, dass es sich lohnt, nach Möglichkeit auf for-Schleifen zu verzichten.

Als nächstes wollen wir einen Vergleich mit Matlab's und Scipy's linprog machen. Eine jeweils ausführliche Dokumentation der Tools ist in [SciLin] für Scipy bzw. in [MatLin] zu finden. Um Matlab-Codes in Python aufzurufen, benötigen wir das matlab.engine Paket. Dieses unterstützt zur Zeit nur Python 3.4 oder älter (die aktuellste Version ist 3.6). Eine Anleitung zur Installation, sowie die gesamte Dokumentation ist unter [MatEn] zu finden. Wir testen wieder mit je 100 Zufallsmatrizen und verschieden großen Werten für $m$ und $n$. Der vollständige Code des Mainfiles ist im Anhang zu finden. Wir erhalten folgende Werte:

| | | Zeit | Iterationen | Wert der Zielfkt |
|---|---|---|---|---|
| $m = 50$ | Simplex | 0.026 | 54.0 | 959.309 |
| $n = 60$ | Scipy | 0.045 | 57.67 | 959.309 |
| iter $= 10$ | Matlab | 0.044 | 51.72 | 959.309 |
| $m = 100$ | Simplex | 0.169 | 189.69 | 482.443 |
| $n = 120$ | Scipy | 0.259 | 193.67 | 482.443 |
| iter $= 10$ | Matlab | 0.123 | 167.88 | 482.443 |
| $m = 200$ | Simplex | 1.343 | 856.18 | 521.204 |
| $n = 240$ | Scipy | 1.537 | 710.34 | 521.204 |
| iter $= 10$ | Matlab | 0.782 | 630.61 | 521.204 |
| $m = 300$ | Simplex | 6.053 | 2067.75 | 820.35 |
| $n = 360$ | Scipy | 5.534 | 1788.16 | 820.35 |
| iter $= 10$ | Matlab | 3.578 | 1593.46 | 820.35 |
| $m = 400$ | Simplex | 27.802 | 4084.97 | 1032.939 |
| $n = 480$ | Scipy | 14.881 | 3734.09 | 1032.939 |
| iter $= 10$ | Matlab | 12.261 | 3311.09 | 1032.939 |

Damit können wir festhalten, dass eine recht einfache Implementierung in Python bei kleinen Problemen sehr gut mit Matlab mithalten kann. Für größere sind die Ergebnisse vorhersehbar. Das ein kommerzielles Programm hier schneller ein Ergebnis liefert, ist nicht überraschend. Sicher ließen sich aber die Werte noch verbessern, wenn man zusätzlich etwa Update-Formeln oder eine bessere Pivot-Strategie verwenden würde. Preprocessing, was in kommerziellen Programmen auch stets angewendet wird, könnte die Rechenzeit ebenfalls verkürzen. Es bleibt jedenfalls festzuhalten, dass Python nicht schlechter ist als Matlab, was wir an den Ergebnissen ablesen können, die Scipy liefert.

### 4.3.2 Branch-and-Bound

Wir wählen wieder Zufallsmatrizen aus, die ein entsprechendes ILP

$$\begin{aligned} &\max c^T x \\ \text{u.d.N. } &Nx \leq b \\ &x \geq 0 \\ &x \text{ ganzzahlig} \end{aligned}$$

mit $c \in \mathbb{R}^m$, $N \in \mathbb{R}^{m \times n}$ und $b \in \mathbb{R}^n$, repräsentieren, und lösen dieses. Zunächst wollen wir testen, welche Reihenfolge der Knotenabarbeitung besser ist (Vorerst ohne ein Initial-Verfahren). Wir erhalten bei hundert verschiedenen Zufallsmatrizen die folgenden Durchschnittswerte, wobei wir maximal 1000 Iterationen zulassen.

| | | Zeit | Iterationen | Wert der Zielfkt | Max Iter |
|---|---|---|---|---|---|
| $m = 5$ | deppfirst | 0.009 | 5.97 | 23.20 | 0 |
| $n = 6$ | bestsolution | 0.015 | 13.85 | 23.20 | 0 |
| $m = 10$ | deepfirst | 0.084 | 62.0 | 84.610 | 0 |
| $n = 12$ | bestsolution | 0.239 | 167.03 | 84.610 | 6 |
| $m = 15$ | deepfirst | 0.122 | 70.30 | 37.136 | 7 |
| $n = 18$ | bestsolution | 0.357 | 205.29 | 37.136 | 20 |
| $m = 20$ | deepfirst | 0.155 | 73.41 | -3.617 | 39 |
| $n = 24$ | bestsolution | 0.492 | 1788.16 | -3.617 | 51 |

Wir sehen also, dass für ein Initial-Verfahren, bei dem eine erste zulässige Lösung gesucht wird, das deepfirst Verfahren vermutlich schneller ist. Ebenfalls sehen wir, dass bereits bei recht kleinen Problemen ($m = 20$, $n = 24$) 1000 Iterationen oft nicht ausreichen (die Durchschnittswerte sind jeweils ohne die Probleme, bei denen das Maximum an Iterationen erreicht wurde). Wir testen nun mit Initial-Verfahren in mehreren Varianten, aber jedes mal mit $m = 20$ und $n = 24$ sowie maximal 1000 Iterationen für das Hauptverfahren.

| Initial | B & B | Zeit | Iterationen | Wert der Zielfkt | Max Iter |
|---|---|---|---|---|---|
| deppfirst (erste zul. Lösung) | bestsolution | 1.070 | 181.56 | 39.472 | 29 |
| bestsolution (erste zul. Lösung) | deppfirst | 0.591 | 157.09 | 39.472 | 28 |
| deepfirst (bis zur 500. It.) | bestsolution | 0.566 | 96.61 | 1.404 | 39 |
| bestsolution (bis zur 500. It.) | deepfirst | 1.306 | 98.14 | 1.404 | 32 |

Im Hinblick, möglichst viele Probleme in kurzer Zeit, sprich mit wenig Iterationen, lösen zu können, scheint die Variante mit Initial-Verfahren bis zur ersten zulässigen Lösung besser zu sein.

Nun wollen wir noch testen, was es für einen Unterschied macht, nach welcher Variable wir verzweigen. Wir wählen die auf den ersten Blick schlechtere Variante, da dies auch unter anderem von Wolsey (vergleiche [Wol98] Kapitel 7) empfohlen wird, und erhalten für $m = 20$, $n = 24$, sowie einer Maximalanzahl an Iterationen von 1000, die folgenden Werte.

| Verzweigung | Zeit | Iterationen | Wert der Zielfkt | Max Iter |
|---|---|---|---|---|
| weitesten Weg von Ganzz | 5.995 | 258.0 | 156.125 | 65 |
| kl Abst zu unterer Gaußklammer | 6.575 | 358.61 | 156.125 | 73 |
| gr Abst zu unterer Gaußklammer | 6.090 | 244.15 | 156.125 | 67 |
| erster mögliche Index | 6.226 | 279.47 | 156.125 | 71 |
| strong Branching | 9.175 | 265.31 | 156.125 | 67 |

Da wir hier kaum eine Aussage treffen können, machen wir den gleichen Test nochmal, dieses mal starten wir jedoch das Initial-Verfahren mit bestsolution und das eigentliche Verfahren mit deepfirst.

| Verzweigung | Zeit | Iterationen | Wert der Zielfkt | Max Iter |
|---|---|---|---|---|
| weitesten Weg von Ganzz | 5.883 | 368.0 | 62.85 | 62 |
| kl Abst zu unterer Gaußklammer | 6.699 | 638.57 | 62.85 | 64 |
| gr Abst zu unterer Gaußklammer | 5.954 | 428.28 | 62.85 | 65 |
| erster mögliche Index | 6.104 | 441.14 | 62.85 | 71 |
| strong Branching | 8.773 | 356.28 | 62.85 | 63 |

Damit scheint bestsolution als Initial-Verfahren zusammen mit deepfirst im eigentlichen Algorithmus sowie die Verzweigungsregel, bei der der Index so gewählt wird, dass die entsprechende Variable am weitesten Weg ist, die beste Kombination zu sein. Das strong

Branching liefert zwar ähnliche Ergebnisse, benötigt aber wesentlich mehr Zeit. Zum Schluss vergleichen wir eben diese mit verschiedenen Werten für $m$ und $n$ mit Matlab und lassen dieses Mal 10.000 Iteration als Maximum zu. Weiterhin nehmen wir die benötigte Zeit und die benötigten Iterationen mit auf, auch wenn das Maximum an Iterationen erreicht wird. Matlab gestatten wir hier ein Maximum an Iterationen von $10^6$.

| | | Zeit | Iterationen | Wert der Zielfkt | Max Iter |
|---|---|---|---|---|---|
| $m = 5$ | B & B | 0.026 | 9.48 | 58.68 | 0 |
| $n = 10$ | Matlab | 0.017 | 61.32 | 58.68 | 0 |
| $m = 10$ | B & B | 0.148 | 141.20 | 159.512 | 0 |
| $n = 12$ | Matlab | 0.021 | 424.39 | 159.512 | 0 |
| $m = 15$ | B & B | 3.363 | 1992.22 | 179.54 | 7 |
| $n = 18$ | Matlab | 0.392 | 7391.86 | 179.54 | 0 |
| $m = 20$ | B & B | 12.709 | 3814.94 | 104.035 | 23 |
| $n = 24$ | Matlab | 1.641 | 36694.58 | 104.035 | 1 |
| $m = 30$ | B & B | 34.218 | 5329.32 | 2.027 | 46 |
| $n = 36$ | Matlab | 70.134 | 896649.49 | 2.027 | 9 |

Wir können festhalten, dass Matlab zwar in der Regel deutlich mehr Knoten untersucht, dieses aber auch deutlich schneller erledigt. Hier zeigt sich, dass selbst bei kleinen Problemen der Aufwand sehr hoch ist, die beweisbar beste Lösung zu finden. Als letzten Test wollen wir sehen, ob es eine Verbesserung bringt, wenn im Laufe des Algorithmus die Regel zur Verzweigung geändert wird. Dazu fügen wir

```
if np.mod(maxiter, 250) == 0:
        variable_search = np.mod(variable_search+1, 3)
```

in den Branch-and-Bound Code ein. Dies ändert nach 250 Iterationen die Verzweigungsregel, lässt aber das strong branching nicht zu. Dies wollen wir hier ausschließen, da die Rechenzeit deutlich höher ist, gleichzeitig aber kaum bessere Ergebnisse liefert. Wir erhalten folgende Ergebnisse:

| | | Zeit | Iterationen | Wert der Zielfkt | Max Iter |
|---|---|---|---|---|---|
| $m = 5$ | B & B | 0.015 | 7.21 | 74.91 | 0 |
| $n = 10$ | Matlab | 0.016 | 26.07 | 74.91 | 0 |
| $m = 10$ | B & B | 0.250 | 86.73 | 164.42 | 0 |
| $n = 12$ | Matlab | 0.044 | 1182.2 | 164.42 | 0 |
| $m = 15$ | B & B | 1.25 | 519.98 | 104.87 | 1 |
| $n = 18$ | Matlab | 0.465 | 13155.38 | 104.87 | 0 |
| $m = 20$ | B & B | 5.944 | 2289.69 | 100.5 | 9 |
| $n = 24$ | Matlab | 3.388 | 50711.70 | 100.5 | 1 |
| $m = 30$ | B & B | 17.990 | 4865.17 | 12.85 | 44 |
| $n = 36$ | Matlab | 16.776 | 229477.76 | 12.85 | 15 |

Was durchaus eine deutliche Verbesserung bewirkt. Eine Implementierung für den dualen Simplex, welcher günstiger wäre als der von uns, würde hier sicher noch bessere Ergebnisse liefern. Damit zeigt sich, dass Python durchaus eine Alternative zu kommerziellen Paketen sein kann. Weiterhin lässt sich festhalten, dass eine Auswahlregel, die im Allgemeinen schnell das optimale Ergebnis liefert, nicht unter den oben genannten

dabei ist. Eine Auswahlregel, die für ein Problem sehr gut erscheint, kann für das nächste wiederum sehr schlecht sein. Als Beispiel seien

$$\begin{aligned} &\max c^T x \\ \text{u.d.N. } &Nx \leq b \\ &x \geq 0 \\ &x \text{ ganzzahlig} \end{aligned}$$

mit

$$c = \begin{pmatrix} 10 \\ 8 \\ 7 \\ 10 \end{pmatrix}, \qquad N = \begin{pmatrix} -9 & 0 & -10 & 5 \\ 5 & 8 & 5 & 7 \\ -3 & -1 & -4 & 1 \\ 7 & 0 & -10 & -1 \\ 10 & 0 & 2 & -2 \end{pmatrix}, \quad \text{und} \quad b = \begin{pmatrix} 0 \\ 3 \\ 49 \\ 78 \\ 70 \end{pmatrix}$$

gegeben. Starten wir hier mit einem deepfirst Initial-Verfahren, welches nach dem Strongbranching verzweigt, und im Anschluss nach der bestsolution Methode und der Verzweigungsregel, dass stets jene Variable gewählt wird, die am weitesten entfernt von der Ganzzahligkeit ist, benötigen wir hier 1008 Iterationen um das Optimum zu finden. Starten wir hingegen mit demselben Initial-Verfahren, verzweigen aber beide Male nach der Regel, die den Index wählt, wo die Variable am Weitesten von der Ganzzahligkeit entfernt ist, benötigen wir nur 12 Iterationen.

# 5 Literaturverzeichnis

## 5.1 Fachbücher und Skripte


[Dom15] **Wolfgang Domschke, Andreas Drexl, Robert Klein, Armin Scholl**, Einführung in Operations Research, neunte Auflage, Springer Gabler 2015

[Due08] **Miriam Dür, Alexander Martin und Stefan Ulbrich**, Skript der Vorlesung Öptimierung I - Einführung in die Optimierung vom Wintersemester 2008/09 an der Technischen Universität Darmstadt, unter:
`http://www.mathematik.tu-darmstadt.de/lehrmaterial/WS2008-2009/Opt_Einf/Skript/skript_Opt1.pdf`
Abfrage am 28. Oktober 2017

[Ham16] **Horst W. Hamacher und Kathrin Klamroth**, Lineare Optimierung und Netzwerkoptimierung - zweisprachige Ausgabe, zweite Auflage, Vieweg 2006.

[Hel14] **Stephan Held**, Skript zur Linearen und Ganzzahligen Optimierung, Forschungsinstitut für Diskrete Mathematik an der Universität Bonn, 2014, unter:
`http://www.or.uni-bonn.de/~held/lpip/1314/skript/Lingo1314.pdf`
Abfrage am 28. Oktober 2017

[Hoo12] **John N. Hooker**, Integrated Methods for Optimization, zweite Auflage, Springer US 2012

[Kor12] **Bernhard Korte, Jens Vygen**, Combinatorial Optimization, Theory and Algorithms, fünfte Auflage, Springer-Verlag Berlin Heidelberg 2012

[Lu15] **Eberhard Luik**, Skript der Vorlesung von Dr. Luik: "Numerik I"vom Wintersemester 2015/16 an der Universität Konstanz, Fachbereich Mathematik und Statistik.

[Mar99] **Richard Kipp Martin**, Large Scale Linear and Integer Optimization: A Unified Approach, erste Auflage, Springer US 1999.

[Sie96] **Gerard Sierksma**, Linear and Integer Programming - Theory and Practice, First Edition, Dekker 1996

[SuMe13] **Leena Suhl, Taïeb Mellouli**, Optimierungssysteme - Modelle, Verfahren, Software, Anwendungen, dritte Auflage, Springer-Verlag Berlin Heidelberg 2013

[The14] **Thomas Theis**, Einstieg in Python, 4. Auflage, Galileo Press, Bonn 2014

[Van14] **Robert J. Vanderbei**, Linear Programming - Foundations and Extensions, Fourth Edition, Springer 2014.

[Wei16] **Michael Weigend**, Python 3 - Lernen und professionell anwenden. Das umfassende Praxisbuch, 6. erweiterte Auflabe, mitp 2016

[Wol98] **Laurence A. Wolsey**, Integer Programming, erste Auflage, Wiley-Interscience 1998

## 5.2 Artikel aus Zeitschriften

[Ach05] **Achterberg, T.; T. Koch und A. Martin**, Branching rules revisited. Operations Research Letters 33, 2005, S. 42 - 54.

[Bel13] **Pietro Belotti, Christian Kirches, Sven Leyffer, Jeff Linderoth, James Luedtke und Ashutosh Mahajan**, Mixed-integer nonlinear optimization, erschienen 2013 in Acta Numerica, p 1–131, unter:
`http://me8xh5ls2s.search.serialssolutions.com/?ctx_ver=Z39.88-2004&ctx_enc=info%3Aofi%2Fenc%3AUTF-8&rfr_id=info%3Asid%2Fsummon.serialssolutions.com&rft_val_fmt=info%3Aofi%2Ffmt%3Akev%3Amtx%3Ajournal&rft.genre=article&rft.atitle=Mixed-integer+nonlinear+optimization&rft.jtitle=ACTA+NUMERICA&rft.au=Belotti%2C+P&rft.au=Kirches%2C+C&rft.au=Leyffer%2C+S&rft.au=Linderoth%2C+J&rft.date=2013&rft.pub=CAMBRIDGE+UNIV+PRESS&rft.issn=0962-4929&rft.eissn=1474-0508&rft.volume=22&rft.spage=1&rft.epage=131&rft_id=info:doi/10.1017%2FS0962492913000032&rft.externalDBID=n%2Fa&rft.externalDocID=000344811600001&paramdict=de-DE`
Abfrage am 28. Oktober 2017

[Bor14] **Karl Heinz Borgwardt**, Wie schnell arbeitet das Simplexverfahren normalerweise? Oder: Das Streben nach (stochastischer) Unabhängigkeit, erschienen 2014 in den DMV-Mitteilungen 2014, Heft 2, Band 22, S. 80 -92, unter:
`https://www.uni-ulm.de/fileadmin/website_uni_ulm/mawi.inst.100/vorlesungen/ss14/Komplexit_des_Simplexverfahrens.pdf`
Abfrage am 28. Oktober 2017

[Dak65] **R. J. Dakin**, A tree-search algorithm for mixed integer programming problems, The Computer Journal, Volume 8, 1965, S. 250–255

[Har73] **Paula Harris**, Pivot selection methods for the Devex LP code, Math. Prog. 5, 1973, Seite 1-28.

[LaDo60] **A. H. Land und A. G. Doig**, An Automatic Method of solving descrete Programming Problems, Econometrica, Volume 28, Number3, 1960, Seite 497 - 516

[KlMi72] **Viktor Klee, George J. Minty**, How Good is the Simplex Algorithm?, O. Shisha, editor, Inequalities, III, Seite 159-175, AcademicPress,NewYork 1972.

[Sch97] **Scholl, A.; G. Krispin, R. Klein und W. Domschke**, Branch-and-Bound – Optimieren auf Bäumen: je beschränkter, desto besser., c't-Magazin für Computer Technik, Heft10, S.336 - 345

[Wol80] **Laurence A. Wolsey**, Heuristic Analysis, Linear Programming and Branch-and-Bound, Mathematical Programming Study 13, 1980, Seite 121-134

## 5.3 Internetquellen

[Bor04] **Karl Heinz Borgwardt**, Die mittlere Schrittzahl beim Simplexverfahren als vorlesbare Vereinfachung vom 29.04.2004 an der Universität Augsburg, Institut für Mathematik, unter:
`https://opus.bibliothek.uni-augsburg.de/opus4/files/53/Simpel.pdf`
Abfrage am 28. Oktober 2017

[MatEn] **Ausführliche Dokumentation über das Matlab Engine Paket für Python**, unter
`https://de.mathworks.com/help/matlab/matlab-engine-for-python.html`
Abfrage am 28. Oktober 2017

[MatLin] **Ausführliche Dokumentation über linprog von Matlab**, unter
`https://de.mathworks.com/help/optim/ug/linprog.html`
Abfrage am 28. Oktober 2017

[Num] **Ausführliche Dokumentation über das Paket numpy**, unter
`http://www.numpy.org/`
Abfrage am 28. Oktober 2017

[Pyth] **Quotes about Python**, unter
`https://www.python.org/about/quotes/`
Abfrage am 28. Oktober 2017

[Sci] **Ausführliche Dokumentation über das Paket scipy**, unter
`https://www.scipy.org/`
Abfrage am 28. Oktober 2017

[SciLin] **Ausführliche Dokumentation über linprog von scipy**, unter
`https://docs.scipy.org/doc/scipy-0.15.1/reference/generated/scipy.optimize.linprog.html`
Abfrage am 28. Oktober 2017

[Wiki] **Wikipedia.de**, Simplex Verfahren, unter:
`https://de.wikipedia.org/wiki/Simplex-Verfahren`
Abfrage am 28. Oktober 2017
Branch-and-Bound, unter:
`https://de.wikipedia.org/wiki/Branch-and-Bound`
Abfrage am 28. Oktober 2017

[Wun96] **Roland Wunderlich** Paralleler und Objektorientierter Simplex-Algorithmus, Dissertation, Berlin 1996, unter:
`https://opus4.kobv.de/opus4-zib/files/538/TR-96-09.pdf`
Abfrage am 28. Oktober 2017

# 6 Anhang

## 6.1 Simplex

```
import numpy as np
import time



#_____________________________________________________________________


def simplexdivision(a,b):
    if a==0 and b==0:
        c = 0
    elif a < 0 and b==0:
        c = -np.inf
    elif a > 0 and b==0:
        c = np.inf
    else:
        c = a/b
    return c

#_____________________________________________________________________

#alternative to simplexdivision

def division( a, b ):
    """
    ignore / 0,
    division( np.array([-1, 0, 1]), np.array([0,0,0]) )
        -> [-inf, 0, inf]
    """
    with np.errstate(divide='ignore', invalid='ignore'):
        a[ abs(a) <= 10**(-10) ] = 0
        b[ abs(b) <= 10**(-10) ] = 0
        c = np.true_divide( a, b )
        c[  np.isnan( c )] = 0  #NaN
    return c

#_____________________________________________________________________

def dsimplex(P, v, b, disp=0, maxiter = 10000):
    """
    dual simplex method to solve Problems after adding a new constraint
    args:       P     the set with the return of the old problem
                v     represent the vector wich represent the new constraint
                b     the new contraint
    opt args:   disp = 0:
                    simplex doesn't show any results
                disp = 1:
```

```
            simplex present some results
returns:    x_opt is the vector of the optimal values
          f_opt is the maximum of c^Tx
          iterations is the number of iterations
          time  represent the the time the algorihtm need
          A     is the initial matrix
          CB and CN are die basic and nonbasic sets from the old one
          x_b   is the x_b from the old one
          z_n   is the z_n from the old one
          c     is the vector which represent the cost function
          b     is the whole vector which represent all constraint
          B     represent the last basic matrix
          """
start = time.time()

c_0 = P['c'].copy()
z_n = P['z_n'].copy()
CB = P['CB'].copy()
CN = P['CN'].copy()
n = np.size(CB)
m = np.size(CN)
CB = np.hstack([CB,n+m+1])
n = n+1
x_b = np.hstack([P['x_b'], -np.dot(v,P['x_opt'])+ b]).copy()
v = np.hstack([v, np.zeros(n-1)]).copy()
e = np.eye(n,1,1-n).copy()
A = np.hstack([np.vstack([P['A'],v]),e]).copy() #define new A
B = np.zeros((n,n)).copy()
index_CB = CB - 1
index_CN = CN - 1
B = A[:, index_CB]
C = A[:, index_CN]
count = 0                    #define counter
x_opt = np.zeros(m).copy()      #define initial x_opt
bound = 0                     #define initial boundness

#solve with dual simplex
solve = solve_dual(C, n, m, B, A, x_b, z_n, CB, CN, count, x_opt, bound, maxiter)

bound = solve['bound']
x_opt = solve['x_opt']
count = solve['iterations']
CN = solve['CN']
CB = solve['CB']
x_b = solve['x_b']
z_n = solve['z_n']
f_opt = np.dot(c_0.T,x_opt)
comptime = (time.time() - start)



if bound == 0:
    if disp == 1:
        print("optimal value of the costfunction:")
        print(f_opt)
        print("-----------------------------------------")
        print("number of iterations:" , count)
        print("-----------------------------------------")
        print("time:" , comptime)
        print("-----------------------------------------")
```

```
        print("_______***end of program***_______________")
    return {'x_opt': x_opt, 'f_opt': f_opt, 'iterations': count, \
            'time': comptime, 'A': A, 'CN': CN, 'CB': CB, 'x_b': x_b, \
            'z_n': z_n, 'c': c_0, 'B': B, 'b': b, 'bound': bound}

elif bound ==2:
    x_opt = 0
    if disp == 1:
        print("allowed range is probably empty")
        print("try it again with less constraints")
        print("can't find global optimum")
        print("_____________________________________________")
        print("_______***end of program***_______________")
    return {'iterations': count, 'f_opt': "no solution found", 'time': comptime, \
            'bound': bound, 'x_opt': x_opt}

elif bound == 3:
    x_opt = 0
    if disp == 1:
        print("maxiter is reached")
    return {'iterations': count, 'f_opt': f_opt, 'time': comptime, 'bound': bound, \
            'x_opt': x_opt}


#_____________________________________________________________________

def simplex(c, N, b, disp=0, maxiter = 10000):
    """
    simplex method to solve problems like max c^Tx s.T. Nx <= b
    args:       c, b has to be 1-d numpy.array, N has to be a 2-d numpy.array
                c: from the costfunction c^Tx
                N: is the matrix from the submitted
                b: is the vector of the inequalities
    opt args:   disp = 0:
                    simplex doesn't show any results
                disp = 1:
                    simplex present some results
    returns:
    if the problem is feasible and bound than it returns the optimum, if not,
    then it return the status unbound or infeasible:
    feasible and bound:
                'feasible': status which represent the feasible of the problem;
                    feasible = 1 means that the problem is primal feasible
                    feasible = 2 means that the problem is dual feasible
                    feasible = 3 means that the problem is wether primal nor dual
                            feasible
                'bound': status of the bound
                    bound = 0 means, that the problem is bound, the algorithm has
                            found a solution
                    bound = 1 menas, that the problem is primal unbound, that
                            means, that the problem ist unbound
                    bound = 2 means, that the problem is dual unbound, that means
                            that the problem is infeasible
                    bound = 3 means, that the maxiter is reached
                'x_opt': x_opt is a 2-d-numpy.array which represent the optimal x
                'f_opt': f_opt is a 2-d-numpy.array which represent the value
                        of the optimal solution
                'iterations': an integer which represent the number of
                            iterations the algorithm need
                'time': the time which the algorithm need
```

```
        'A': a 2-d-array which represent the matrix A = [N,B]
        'CN': a 1-d-array; CN represent the final set of nonbasic indices
        'CB': a 1-d-array; CB represent the final set of basic indices
        'x_b': a 2-d-array, x_b is the final vector of x_b from the algorithm
        'z_n': a 2-d-array, z_n is the final vector of z_n from the algorithm
        'c': c the input 2-d-array c
        'b': b the input 2-d-array b
unfeasible or unbound:
        if the problem is unbound or infeasible you get
        'iterations': the number of iterations the algorithm need
        'time': the time which the algorithm need
        'f_opt': you get the status: "no solution found"
        if bound = 1 you get the status "the problem seems to be unbound"
           "try it again with more constraints"
        if bound = 2 you geht the status "allowed range is probably empty"
           "try it again with less constraints"
method:    if feasible = 1, the algorithm try to solve the problem with
           the primal simplex method (solve_primal)
        if feasible = 2, the algorithm try to solve the problem with
           the dual simplex method (solve_dual)
        if feasibel = 3 the algorithm substitute the probel into a
           primal/dual feasible problem and try to solve the problem
           with the primal/dual simplex method than it try to solve
           the originally problem wiht the primal/dual simplex
           mehtod
packages:  numpy as np
        time
"""
start = time.time()

#set initial values
C = N.copy()                #define nonbasicmatrix C instead of N
[n,m] = np.shape(C)         #check the dimension of N
B = np.eye(n)               #define basicmatrix B
A = np.hstack([C,B])        #define matrix A
x_b = b                     #define basic x
z_n = -c
#define basic and nonbasic
CB = np.arange(m+1,m+n+1)     #basic
CN = np.arange(1,m+1)         #nonbasic

count = 0                   #define counter
x_opt = np.zeros(m)         #define initial x_opt
bound = 0                   #define initial boundness
feasible = 0                #define initial feasible


# if the problem is primal feasible, set feasible to 1, if it is dual
#feasible set feasible to 2, else set feasible to 3
#check feasible
if np.where(x_b < -10**(-10), True, False).any() == False:
   feasible = 1
elif np.where(z_n < -10**(-10), True, False).any() == False:
   feasible = 2
else:
   feasible = 3


#solve with simplex
if feasible == 1:
   solve = solve_primal(C, n, m, B, A, x_b, z_n, CB, CN, count, x_opt, bound, \
```

```
                    maxiter)

elif feasible == 2:
    solve = solve_dual(C, n, m, B, A, x_b, z_n, CB, CN, count, x_opt, bound, \
                    maxiter)
elif feasible == 3:
    if disp == 1:
        print("-------------------------------------------")
        print("the problem is not primal and not dual feasible, \
            try a two phase algorithm")
        print("-------------------------------------------")
    #substitute
    x_b_sub = np.ones(len(b))
    solve_sub = solve_primal(C, n, m, B, A, x_b_sub, z_n, CB, CN, count, x_opt,\
                        bound, maxiter)
    count = solve_sub['iterations']

    if solve_sub['bound'] == 1:
        bound = 1
    elif solve_sub['bound'] == 0:
        CN = solve_sub['CN']
        CB = solve_sub['CB']
        index_CB = CB - 1
        index_CN = CN - 1
        B = np.take(A, index_CB, axis = 1)
        C = np.take(A, index_CN, axis = 1)
        z_n = solve_sub['z_n']
        x_b = np.linalg.solve(B, b)
        solve = solve_dual(C, n, m, B, A, x_b, z_n, CB, CN, count, x_opt,\
                        bound, maxiter)
    elif solve_sub['bound'] == 3:
        bound = solve_sub['bound']

#define results from simplex
if feasible != 3:
    bound = solve['bound']
    x_opt = solve['x_opt']
    count = solve['iterations']
    CN = solve['CN']
    CB = solve['CB']
    x_b = solve['x_b']
    z_n = solve['z_n']
    f_opt = np.dot(c.T,x_opt)
elif feasible == 3 and bound == 0:
    bound = solve['bound']
    x_opt = solve['x_opt']
    count = solve['iterations']
    CN = solve['CN']
    CB = solve['CB']
    x_b = solve['x_b']
    z_n = solve['z_n']
    f_opt = np.dot(c.T,x_opt)


comptime = (time.time() - start)

#output after checking bound
if bound == 0:
    if disp == 1:
        print("optimal value of the costfunction:")
        print(f_opt)
```

```
            print("-------------------------------------------")
            print("number of iterations:" , count)
            print("-------------------------------------------")
            print("time:" , comptime)
            print("-------------------------------------------")
            print("________***end of program***________________")
        return {'feasible': feasible, 'bound': bound, 'x_opt': x_opt, \
                'f_opt': f_opt, 'iterations': count, 'time': comptime, \
                'A': A, 'CN': CN, 'CB': CB, 'x_b': x_b, 'z_n': z_n, 'c': c, \
                'b': b}

    elif bound == 1:
        if disp == 1:
            print("the problem seems to be unbound")
            print("try it again with more constraints")
            print("can't find global optimum")
            print("_____________________________________________")
            print("________***end of program***________________")
        x_opt = 0
        return {'iterations': count, 'f_opt': "no solution found", 'time': comptime,\
                'bound': bound, 'x_opt': x_opt}

    elif bound == 2:
        if disp == 1:
            print("allowed range is probably empty")
            print("try it again with less constraints")
            print("can't find global optimum")
            print("_____________________________________________")
            print("________***end of program***________________")
        x_opt = 0
        return {'iterations': count, 'f_opt': "no solution found", 'time': comptime, \
        'bound': bound, 'x_opt': x_opt}

    elif bound == 3:
        if disp == 1:
            print("maxiter is reached")
            print("can't find global optimum")
            print("_____________________________________________")
            print("________***end of program***________________")
        x_opt = 0
        return {'iterations': count, 'f_opt': "no solution found", 'time': comptime, \
                'bound': bound, 'x_opt': x_opt}


#_____________________________________________________________


def solve_primal(C, n, m, B, A, x_b, z_n, CB, CN, count, x_opt, bound, maxiter):
    """
    simplex method to solve a primal feasible problem like max c^Tx s.T. Ax = b
    args:       C: copy from N, the matrix from the submitted
                n, m: are the integers which represent the size from A
                B: represent the basic matrix
                A: represent the whole matrix A = [N,B]
                x_b: a 2-d-array, vector of x_b from the algorithm, normaly x_b = b
                z_n: a 2-d-array, vector of z_n from the algorithm, normaly z_n = -c
                CN: a 1-d-array; CN represent the set of nonbasic indices
                CB: a 1-d-array; CB represent the set of basic indices
                count: an integer number wich counts the number of iterations
                x_opt: a 2-d-array, normaly at the beginning it is a zero-vector
```

```
        bound: the inital bound
returns:
if the problem is bound than it returns the optimum, if not, then it returns
the status that the problem is unbound:
        'bound': status of the bound
            bound = 0 means, that the problem is bound, the algorithm has
                found a solution
            bound = 1 menas, that the problem is primal unbound, that
                means, that the problem is unbound
            bound = 3 means, that maxiter is reached
        'x_opt': x_opt is a 2-d-numpy.array which represent the optimal x
        'iterations': an integer which represent the number of
                iterations the algorithm need
        'CN': a 1-d-array; CN represent the final set of nonbasic indices
        'CB': a 1-d-array; CB represent the final set of basic indices
        'x_b': a 2-d-array, x_b is the final vector of x_b from the algorithm
        'z_n': a 2-d-array, z_n is the final vector of z_n from the algorithm

unbound:
        if the problem is unbound you get the same output
method:
        the algorithm need div0(a,b) because of the convention 0/0 = 0
packages:
        numpy as np
"""
ma_test = np.ma.masked_where(z_n >= -10**(-10), z_n, copy=False)
while ma_test.count() > 0 and count <= maxiter:
    #select entering variable: (choose j like in Bland)
    test = np.ma.where(ma_test == ma_test.min())
    j = np.min(CN[test])
    j_search = np.where(j == CN)[0][0]

    #compute kth unit vector, where k is the position of j in CN
    k = j_search
    e_k = np.eye(m,1,1-(k+1))
    e_k.shape = (m)

    #compute primal step direction deltax_b (deltax_b = B^(-1)Ne_k)
    C_tilde = np.dot(C,e_k)
    deltax_b = np.linalg.solve(B,C_tilde)

    #compute primal step lenght and choose leaving variable
    T = division(deltax_b,x_b)

    if np.where(T <= 10**(-10), True, False).all():
        count = count -1
        bound =  1
        break

    #select leaving variable like in Bland'sche pivotrule
    T_bar = np.argwhere(T == np.amax(T))
    i = np.min(CB[T_bar])
    Max_T_bar = np.where(i == CB)[0][0]
    t = 1/(np.amax(T))

    #compute lth unit vector, where l is the position of i in CB
    l = Max_T_bar
    e_l = np.eye(n,1,1-(l+1))
    e_l.shape = (n)

    #compute dual step direction deltaz_n (deltaz_n = -(B^(-1)N)^Te_l)
```

```
        v = np.linalg.solve(B.T,e_l)
        deltaz_n = np.dot(-C.T,v)

        #compute dual step length
        s = z_n[j_search]/deltaz_n[j_search]

        #update
        x_b = (x_b - t * deltax_b) + t * e_l
        z_n = (z_n - s * deltaz_n) + s * e_k
        CB[l] = j
        CN[k] = i
        B[:,l] = A[:, j-1]
        C[:,k] = A[:, i-1]
        count += 1
        ma_test = np.ma.masked_where(z_n >= -10**(-10), z_n, copy=False)

    if count == maxiter:
        bound = 3
    for i in range(0,m):
        if i+1 in CB:
            x_opt[i] = x_b[np.where( CB == i+1 )]
        else:
            x_opt[i] = 0

    return {'bound': bound, 'x_opt': x_opt, 'iterations': count, 'CN': CN, \
            'CB': CB, 'x_b': x_b, 'z_n': z_n}

#_____________________________________________________________________

def solve_dual(C, n, m, B, A, x_b, z_n, CB, CN, count, x_opt, bound, maxiter):
    """
    simplex method to solve a dual feasible problem like max c^Tx s.T. Ax = b
    args:       C: copy from N, the matrix from the submitted
                n, m: are the integers which represent the size from A
                B: represent the basic matrix
                A: represent the whole matrix A = [N,B]
                x_b: a 2-d-array, vector of x_b from the algorithm, normaly x_b = b
                z_n: a 2-d-array, vector of z_n from the algorithm, normaly z_n = -c
                CN: a 1-d-array; CN represent the set of nonbasic indices
                CB: a 1-d-array; CB represent the set of basic indices
                count: an integer number wich counts the number of iterations
                x_opt: a 2-d-array, normaly at the beginning it is a zero-vector
                bound: the inital bound
    returns:
    if the problem is bound than it returns the optimum, if not, then it returns
    the status that the problem is unbound:
                'bound': status of the bound
                    bound = 0 means, that the problem is bound, the algorithm has
                            found a solution
                    bound = 2 menas, that the problem is dual unbound, that
                            means, that the problem is infeasible
                    bound = 3 means, that maxiter is reached
                'x_opt': x_opt is a 2-d-numpy.array which represent the optimal x
                'iterations': an integer which represent the number of
                            iterations the algorithm need
                'CN': a 1-d-array; CN represent the final set of nonbasic indices
                'CB': a 1-d-array; CB represent the final set of basic indices
                'x_b': a 2-d-array, x_b is the final vector of x_b from the algorithm
                'z_n': a 2-d-array, z_n is the final vector of z_n from the algorithm

    unbound:
```

```
        if the problem is unbound you get the same output
method:
        the algorithm need div0(a,b) because of the convention 0/0 = 0
packages:
        numpy as np
"""


ma_test=np.ma.masked_where(x_b >= -10**(-10), x_b, copy=False)
while ma_test.count() > 0 and count <= maxiter:
    #select entering variable: (choose p like in Bland)
    test = np.ma.where(ma_test == ma_test.min())
    p = np.max(CB[test])
    p_search = np.where(p == CB)[0][0]

    #compute kth unit vector, where k is the position of j in CN
    k = p_search
    e_k = np.eye(n,1,1-(k+1))
    e_k.shape = (n)

    #compute dual step direction deltaz_n (deltaz_n = -(B^(1)N).Te_i)
    v = np.linalg.solve(B.T,e_k)
    deltaz_n = np.dot(-C.T,v)

    #compute primal step lenght and choose leaving variable
    S = division(deltaz_n,z_n)

    if np.where(S <= 10**(-10), True, False).all():
        count = count -1
        bound =  2
        break

    #select leaving variable like in Bland'sche pivotrule
    S_bar = np.argwhere(S == np.amax(S))
    j = np.max(CN[S_bar])
    Max_S_bar = np.where(j == CN)[0][0]
    s = 1/(np.amax(S))

    #compute lth unit vector, where l is the position of i in CB
    l = Max_S_bar
    e_l = np.eye(m,1,1-(l+1))
    e_l.shape = (m)

    #compute primal step direction deltax_b (deltax_b = B^(-1)Ne_j])
    C_tilde = np.dot(C,e_l)
    deltax_b = np.linalg.solve(B,C_tilde)

    #compute dual step length
    t = x_b[p_search]/deltax_b[p_search]

    #update
    x_b = (x_b - t * deltax_b) + t * e_k
    z_n = (z_n - s * deltaz_n) + s * e_l
    CB[k] = j
    CN[l] = p
    B[:,k] = A[:, j-1]
    C[:,l] = A[:, p-1]
    count += 1
    ma_test=np.ma.masked_where(x_b >= -10**(-10), x_b, copy=False)

if count == maxiter:
```

```
        bound = 3
    for i in range(0,m):
        if i+1 in CB:
            x_opt[i] = x_b[np.where( CB == i+1 )]
        else:
            x_opt[i] = 0

    return {'bound': bound, 'x_opt': x_opt, 'iterations': count, 'CN': CN, \
            'CB': CB, 'x_b': x_b, 'z_n': z_n}

#_____________________________________________________________________________
```

## 6.2 Main File Simplex

### 6.2.1 Python

```
from IPython import get_ipython
get_ipython().magic('reset -sf')


import time
import numpy as np
from scipy.optimize import linprog
from simplex import simplex
import matlab.engine
eng = matlab.engine.start_matlab()

import random
random.seed()


#Test it very often
m = 200           #choose the dimension of the unknown
n = 240           #choose the dimension of the contraints
integer = 10      #choose how big the integers

ZeitSimplex = []
ZeitLinprog = []
ZeitMatlab = []
IterSimplex = []
IterLinprog = []
IterMatlab = []
Vgl_f_opt_simplex = []
Vgl_f_opt_linprog = []
Vgl_f_opt_matlab = []
primal = 0
dual = 0
nothing = 0
max_iter = 0

unbound = 0
empty = 0
allright = 0

k = 0
while k < 100:

    #define random number to choose the problem
    randomnumber = random.randint(0,2)

    #primal
    if randomnumber == 0:
        #define a primal feasible problem
        primal = primal + 1
        c = np.zeros(m)
        for i in range(0,m):
            c[i] = random.randint(-integer,integer)

        b = np.zeros(n)
        for i in range(1,n):
            b[i] = random.randint(0,10*integer)
```

```
    N = np.zeros((n,m))
    for i in range(0,n):
        for j in range(0,m):
            N[i][j] = random.randint(-integer,integer)

#dual
if randomnumber == 1:
    #define a dual feasible problem
    dual = dual + 1
    c = np.zeros(n)
    for i in range(0,n):
        c[i] = random.randint(-integer,0)

    b = np.zeros(m)
    for i in range(1,m):
        b[i] = random.randint(-integer,10*integer)

    N = np.zeros((m,n))
    for i in range(0,m):
        for j in range(0,n):
            N[i][j] = random.randint(-integer,integer)

#both
if randomnumber == 2:
    #define a not primal feasible and not dual feasible problem
    nothing = nothing +1
    c = np.zeros(n)
    for i in range(0,n):
        c[i] = random.randint(-integer,integer)

    b = np.zeros(m)
    for i in range(1,m):
        b[i] = random.randint(-integer,10*integer)

    N = np.zeros((m,n))
    for i in range(0,m):
        for j in range(0,n):
            N[i][j] = random.randint(-integer,integer)

#save the linear program to compare it with matlab
np.savetxt("c_general.txt", c, fmt="%2.3f", delimiter=",")
np.savetxt("b_general.txt", b, fmt="%2.3f", delimiter=",")
np.savetxt("N_general.txt", N, fmt="%2.3f", delimiter=",")

#solve with simplex
P = simplex(c,N,b)

#transform into a minimize problem
c_scipy = -1*c
b_scipy = b
N_scipy = N

#solve with linprog from scipy
start = time.time()
P_scipy = linprog(c_scipy, A_ub = N_scipy, b_ub = b_scipy, \
            options={"maxiter": 10000 })
comptime = (time.time() - start)

#solve with matlab
[f_opt_matlab, Iter_matlab, exitflag, time_matlab] = \
      eng.comparison_matlab_python_simplex(nargout = 4)
```

```
    if P['iterations'] > 9999:
        max_iter += 1


    #save time, iter, f_opt to compare the values
    if P_scipy['status']==0 and P['iterations'] <= 9999 and exitflag == 1:
        allright += 1
        ZeitSimplex.append(P['time'])
        ZeitLinprog.append(comptime)
        ZeitMatlab.append(time_matlab)
        IterSimplex.append(P['iterations'])
        IterLinprog.append(P_scipy['nit'])
        IterMatlab.append(Iter_matlab)
        Vgl_f_opt_simplex.append(P['f_opt'])
        Vgl_f_opt_linprog.append(-1*P_scipy['fun'])
        Vgl_f_opt_matlab.append(f_opt_matlab)

    #check if solution found
    if P_scipy['status'] == 3 and P['bound']==1 and exitflag == -3:
        unbound += 1

    if P_scipy['status'] == 2 and P['bound']==2 and exitflag == -2:
        empty += 1



    k += 1
    print(k)

#display the meanvalue of the saved figures
print("-----------------------------------------------------------------------")
print("arithmetic mean of time by using simplex:" , np.mean(ZeitSimplex))
print("arithmetic mean of time by using linprog:" , np.mean(ZeitLinprog))
print("arithmetic mean of time by using matlab: " ,  np.mean(ZeitMatlab))
print("-----------------------------------------------------------------------")
print("arithmetic mean of iterations by using simplex:" , np.mean(IterSimplex))
print("arithmetic mean of iterations by using linprog:" , np.mean(IterLinprog))
print("arithmetic mean of iterations by using matlab: " , np.mean(IterMatlab))
print("-----------------------------------------------------------------------")
print("arithmetic mean of f_opt by using simplex", np.mean(Vgl_f_opt_simplex))
print("arithmetic mean of f_opt by using linprog", np.mean(Vgl_f_opt_linprog))
print("arithmetic mean of f_opt by using matlab ", np.mean(Vgl_f_opt_matlab))
print("-----------------------------------------------------------------------")
print("number of problems which both found an opt          ", allright)
print("number of problems which the problems are unbound    ", unbound)
print("number of problems which the allowed space are empty", empty)
print("number of problems which the maxiter is reached      ", max_iter)
print("-----------------------------------------------------------------------")
```

### 6.2.2 Matlab

```
function [f_opt, iter, exitflag, time] = comparison_matlab_python_simplex()

% c : vector with coefficients of the linear objective function; because of
% Matlab minimize the Problem, we multply it with -1
c = -1*load('c_general.txt');
% N : matrix with coefficients of the linear inequality constraints
N = load('N_general.txt');
% b : vector for linear inequality constraints
b = load('b_general.txt');
% Aineq together with bineq form the inequality constraints
% Aineq*x=bineq .
% Since there are no equality constraints Aeq*x=beq , Aeq and beq do not
% have to be used .
% lb : lower bound constraints (no upper bound constraints -> set ub =[]
lb = zeros (length(c),1) ;
% Options can be chosen manually :
options = optimoptions ( @linprog , 'Algorithm' , 'simplex' , 'Diagnostics' ,...
    'off' , 'Display' , 'final' , 'MaxIter' ,100000) ;
% x: solution x
% fval : value of the objective function at the solution x
% exitflag : informs about the reason for termination
% output : Gives information about the optimization process e . g . the number
% of iterations , the used algorithm or an exit message .
tic;
[x , fval , exitflag , output , lambda ] = linprog ( c , N , b , [ ] , [ ] ,                    ...
    lb , [ ] , [] , options);


%output
f_opt = -fval;
iter = output.iterations;
%exitflag = exitflag;
time = toc;
```

## 6.3 Branch-and-Bound

```
import time
import numpy as np
from simplex import simplex
from simplex import dsimplex


#_____________________________________________________________________

def isinteger(x):
    """
    verifies elementwise to an integer
    args:       an array
    returns:    a mask array
    """
    x_1 = np.around(x, decimals = 6)
    y = np.mod(x_1,1)
    ma = np.ma.masked_where(abs(y) <= 10**(-6), y, copy=False)
    return ma

#_____________________________________________________________________

def strong_branching(P, ma, c):
    """
    verifies the indices which have the minimal opt
    args:       P: solved LP
                ma: masked array which represent the integer test
                c: vector of the costfunction
    returns:    the index which have the minimal opt
    method:     dsimplex
    packages:   numpy as np
    """
    index0 = ma.nonzero()
    index1 = index0[0].astype(int)
    len_index = len(index1)
    len_c = len(c)
    Z = np.vstack([index1, np.zeros(len_index), np.zeros(len_index)])

    for i in range(0,len_index):
        l = Z[0][i]
        l = int(l)

        input_v_1 = np.eye(len_c,1,1-(l+1))
        input_v_1.shape = (len_c)
        input_b_1 = np.floor(P['x_opt'][i])

        input_v_2 = -1*input_v_1
        input_b_2 = -1*(input_b_1 + 1)

        Node_1 = dsimplex(P, input_v_1, input_b_1, maxiter = 10)
        bound_1 = Node_1['bound']
        f_opt_1 = Node_1['f_opt']

        if bound_1 == 2:
            Z[1][i] = np.nan
        elif isinteger(Node_1['x_opt']).count() == 0:
            z = int(Z[0][i])
```

```
            return z
            break
        else:
            Z[1][i] = f_opt_1

        Node_2 = dsimplex(P, input_v_2, input_b_2, maxiter = 10)
        bound_2 = Node_2['bound']
        f_opt_2 = Node_2['f_opt']

        if bound_2 == 2:
            Z[2][i] = np.nan
        elif isinteger(Node_2['x_opt']).count() == 0:
            z = int(Z[0][i])
            return z
            break
        else:
            Z[2][i] = f_opt_2

    if np.isnan(Z[1]).all():
        z = int(Z[0][0])
    elif np.isnan(Z[2]).all():
        z = 0
    else:
        z_1 = np.nanargmin(Z[1])
        z_2 = np.nanargmin(Z[2])
        if Z[1][z_1] >= Z[1][z_2]:
            z = int(Z[0][z_1])
        else:
            z = int(Z[0][z_2])
    return z


#_____________________________________________________________________

def initial_b_and_b(c, N, b, disp=0, variable_search = 0, initial = 'deepfirst',\
                    maxiter = 10**(3)):
    """
    branch and bound method to find a first feasible solution
    args:       c, b has to be 1-d-numpy.array, N has to be a 2-d-numpy.array
                c: from the costfunction c^Tx
                N: is the matrix from the submitted
    opt args:   disp = 0:
                    initial_b_and_b doesn't show any results
                disp = 1:
                    initial_b_and_b present some results
                variable_search =
                    0: choose the index which the variable is furthest away from integer
                    1: choose the first minimal index
                    2: choose the first maximal index
                    3: choose the first possible index
                    4: choose with strong branching
                initial =
                    'deepfirst': try to find an integer solution with deepfirst method
                    'bestsolution': try to find an integer solution with bestsolution
                    method
                maxiter
                    choose an integer number
    returns:    x_opt is the vector of the optimal values
                f_opt is the maximum of c^Tx
                iterations is the number of iterations/nodes
                time  represent the the time the algorihtm need
```

```
            bound =
            0: the problem is bound, the algorithm has found a solution
            1: the problem is primal unbound
            2: the the problem is dual unbound, -> the problem is infeasible
method:     simplex
            dsimplex
packages:   numpy as np
            time
"""
if initial != 'deepfirst' and initial != 'bestsolution':
    print('wrong choice for initial')
    print('choose deepfirst or bestsolution')
    return{}

start = time.time()
count = 0
f_opt = np.array([-1* np.inf])
x_opt = 0
index =  set([0])
finish = set()
f_o = np.array([-1* np.inf])
bound = 0
integerfound = False

Node = simplex(c, N, b)
pos = np.array([[0],[0],[0],[Node],[Node['f_opt']]])

if Node['bound'] == 1:
    comptime = (time.time() - start)
    bound = 1
    if disp == 1:
        print("the problem seems to be unbound")
        print("try it again with more constraints")
        print("can't find global optimum")
        print("_____________________________________________")
        print("________***end of program***_________________")
    return {'time': comptime, 'iterations': Node['iterations'], 'f_opt': f_opt, \
            'bound': bound}

if Node['bound'] == 2:
    comptime = (time.time() - start)
    bound = 2
    if disp == 1:
        print("allowed range is probably empty")
        print("try it again with less constraints")
        print("can't find global optimum")
        print("_____________________________________________")
        print("________***end of program***_________________")
    return {'time': comptime, 'iterations': Node['iterations'], 'f_opt': f_opt, \
            'bound': bound}

Node = simplex(c, N, b)
#define position
pos = np.array([[0],[0],[0],[Node],[Node['f_opt']]])

while index != finish and integerfound == False:
#while index != finish and count <= maxiter:
    #select k
    #bestsolution choose the maximum local upper bound, deepfirst choose
    #the most deep and left node first
    if initial == 'bestsolution':
```

```
    axis = pos[4].astype(float)
    if np.floor(np.nanmax(axis)) == f_opt:
        break
    level_search = np.nanargmax(axis)
elif initial == 'deepfirst':
    level_search = np.argmax(pos[1])
k = int(pos[0][level_search])

count = int(np.max(pos[0]))
if count == maxiter:
    break

#bounding
f_o = pos[4][k]
integer_test = isinteger(pos[3][k]['x_opt'])

#fathomed
if f_o <= f_opt:
    pos[1][k] = 0
    pos[3][k] = 0
    pos[4][k] = np.nan
    index.discard(k)
elif f_o > f_opt and integer_test.count() == 0:
    f_opt = np.around(f_o, decimals = 6)
    integerfound =True
    x_opt = np.around(pos[3][k]['x_opt'], decimals=12)
    pos[1][k] = 0
    pos[3][k] = 0
    pos[4][k] = np.nan
    index.discard(k)
elif pos[3][k]['bound'] == 2:
    pos[1][k] = 0
    pos[3][k] = 0
    pos[4][k] = 0
    index.discard(k)
    print('ja du brauchst denn fall')

#branching
else:
    #choose an index
    if variable_search == 0:
        #choose the index which the variable is furthest away from integer
        i_search = min(np.ma.flatnotmasked_edges(integer_test-0.5))
    elif variable_search == 1:
        #choose the first minimal index
        i_search = np.ma.where(integer_test == integer_test.min())[0][0]
    elif variable_search == 2:
        #choose the first maximal index
        i_search = np.ma.where(integer_test == integer_test.max())[0][0]
    elif variable_search == 3:
        #choose the first possible index
        i_search = min(np.ma.flatnotmasked_edges(integer_test))
    elif variable_search == 4:
        #choose with strong branching
        if integer_test.count() == 1:
            i_search = min(np.ma.flatnotmasked_edges(integer_test))
        else:
            i_search = strong_branching(pos[3][k], integer_test, c)

    #define new nodes
    max_index = max(pos[0])
```

```
            l = i_search

            input_v_1 = np.eye(len(c),1,1-(l+1))
            input_v_1.shape = (len(c))
            input_b_1 = np.floor(pos[3][k]['x_opt'][i_search])
            Node_1 = dsimplex(pos[3][k], input_v_1, input_b_1, maxiter = 1000)
            if Node_1['bound'] == 0:
                if isinteger(Node_1['x_opt']).count() == 0 and Node_1['f_opt'] > f_opt:
                    f_opt = np.around(Node_1['f_opt'], decimals = 6)
                    integerfound =True
                    x_opt = np.around(Node_1['x_opt'], decimals=12)
                    add_1 = np.array([[(max_index)+1], [0], [0], [0], [np.nan]])
                else:
                    add_1 = np.array([[(max_index)+1], [pos[1][k] + 1], [0], [Node_1], \
                                [Node_1['f_opt']]])
                    index.add((max_index)+1)
            elif Node_1['bound'] == 3:
                variable_search = np.mod(variable_search+1, 1)
            else:
                add_1 = np.array([[(max_index)+1], [0], [0], [0], [np.nan]])

            input_v_2 = -1*input_v_1
            input_b_2 = -1*(input_b_1 + 1)
            Node_2 = dsimplex(pos[3][k], input_v_2, input_b_2, maxiter = 1000)
            if Node_2['bound'] == 0:
                if isinteger(Node_2['x_opt']).count() == 0 and Node_2['f_opt'] > f_opt:
                    f_opt = np.around(Node_2['f_opt'], decimals = 6)
                    integerfound =True
                    x_opt = np.around(Node_2['x_opt'], decimals=12)
                    add_2 = np.array([[(max_index)+2], [1], [0], [0], [np.nan]])
                else:
                    add_2 = np.array([[(max_index)+2], [pos[1][k] + 1], [1], [Node_2], \
                                [Node_2['f_opt']]])
                    index.add((max_index)+2)
            elif Node_2['bound'] == 3:
                variable_search = np.mod(variable_search+1, 1)
            else:
                add_2 = np.array([[(max_index)+2], [0], [1], [0], [np.nan]])

            pos = np.hstack([pos,add_1,add_2])
            index.discard(k)
            pos[1][k] = 0
            pos[3][k] = 0
            pos[4][k] = np.nan

    comptime = (time.time() - start)
    return {'time': comptime, 'iterations': count, 'f_opt': f_opt, 'x_opt': x_opt, \
            'bound': bound}


#_____________________________________________________________________

def b_and_b(c, N, b, disp=0, variable_search = 0, initial = 'deepfirst', \
            nodesearch = 'bestsolution', maxiter = 10**(4)):
    """
    branch and bound method to find the best integer solution
    args:       c, b has to be 1-d-numpy.array, N has to be a 2-d-numpy.array
                c: from the costfunction c^Tx
                N: is the matrix from the submitted
    opt args:   disp = 0:
```

```
            initial_b_and_b doesn't show any results
        disp = 1:
            initial_b_and_b present some results
        variable_search =
            0: choose the index which the variable is furthest away from integer
            1: choose the first minimal index
            2: choose the first maximal index
            3: choose the first possible index
            4: choose with strong branching
        initial =
            'deepfirst': try to find an integer solution with deepfirst method
            'bestsolution': try to find an integer solution with bestsolution method
        nodesearch =
            'deepfirst': try to find an integer solution with deepfirst method
            'bestsolution': try to find an integer solution with bestsolution method
        maxiter
            choose an integer number
returns:    x_opt is the vector of the optimal values
        f_opt is the maximum of c^Tx
        iterations is the number of iterations/nodes
        time  represent the the time the algorihtm need
        bound =
            0: the problem is bound, the algorithm has found a solution
            1: the problem is primal unbound
            2: the the problem is dual unbound, -> the problem is infeasible
            3: maxiter is reached
            4: the problem has no integer solution
method:     simplex
        dsimplex
packages:   numpy as np
        time
"""
start = time.time()
count = 0
f_opt = np.array([-1* np.inf])
x_opt = 0
index =  set([0])
finish = set()
f_o = np.array([-1* np.inf])
bound = 0

if initial == 'deepfirst':
    P_initial = initial_b_and_b(c, N, b, disp = 0, variable_search = 0, \
                            initial = 'deepfirst', maxiter = 1000)

elif initial == 'bestsolution':
    P_initial = initial_b_and_b(c, N, b, disp = 0, variable_search = 0, \
                            initial = 'bestsolution', maxiter = 1000)

elif initial != 'deepfirst' and initial != 'bestsolution' and initial != 0:
    print('wrong choice for initial')
    print('choose deepfirst or bestsolution')
    return{}

if initial == 0:
    P_initial = simplex(c, N, b)

if P_initial['bound'] == 1:
    comptime = (time.time() - start)
    bound = 1
    if disp == 1:
```

```
        print("the problem seems to be unbound")
        print("try it again with more constraints")
        print("can't find global optimum")
        print("_____________________________________________")
        print("________***end of program***_________________")
    return {'time': comptime, 'iterations': P_initial['iterations'], 'f_opt': f_opt, \
            'bound': bound}

if P_initial['bound'] == 2:
    comptime = (time.time() - start)
    bound = 2
    if disp == 1:
        print("allowed range is probably empty")
        print("try it again with less constraints")
        print("can't find global optimum")
        print("_____________________________________________")
        print("________***end of program***_________________")
    return {'time': comptime, 'iterations': P_initial['iterations'], 'f_opt': f_opt, \
            'bound': bound}

if P_initial['bound'] == 4:
    comptime = (time.time() - start)
    bound = 4
    if disp == 1:
        print("the problem has no integer solution")
        print("--------------------------------------------")
        print("number of iterations:" , count)
        print("--------------------------------------------")
        print("time:" , comptime)
        print("--------------------------------------------")
        print("________***end of program***_________________")
    return {'time': comptime, 'iterations': P_initial['iterations'], 'f_opt': f_opt, \
            'x_opt': x_opt, 'bound': bound}



Node = simplex(c, N, b)
f_opt = P_initial['f_opt']
x_opt = P_initial['x_opt']
#define position
pos = np.array([[0],[0],[0],[Node],[Node['f_opt']]])

while index != finish:
    #select k
    #bestsolution choose the maximum local upper bound, deepfirst choose
    #the most deep and left node first
    if nodesearch == 'bestsolution':
        axis = pos[4].astype(float)
        if np.nanmax(axis) < f_opt:
            break
        level_search = np.nanargmax(axis)
    elif nodesearch == 'deepfirst':
        level_search = np.argmax(pos[1])
    if np.nanmax(pos[4].astype(float)) < f_opt:
        break
    k = int(pos[0][level_search])

    #test
    if np.mod(maxiter, 250) == 0:
        variable_search = np.mod(variable_search+1, 3)
```

```
#bounding
f_o = pos[4][k]
integer_test = isinteger(pos[3][k]['x_opt'])
count = int(np.max(pos[0]))
if count == maxiter:
    bound = 3
    break

#fathomed
if f_o <= f_opt:
    pos[1][k] = 0
    pos[3][k] = 0
    pos[4][k] = np.nan
    index.discard(k)
elif f_o > f_opt and integer_test.count() == 0:
    f_opt = np.around(f_o, decimals = 6)
    x_opt = np.around(pos[3][k]['x_opt'], decimals=6)
    pos[1][k] = 0
    pos[3][k] = 0
    pos[4][k] = np.nan
    index.discard(k)
elif pos[3][k]['bound'] == 2:
    pos[1][k] = 0
    pos[3][k] = 0
    pos[4][k] = np.nan
    index.discard(k)
    print('ja du brauchst denn fall')
#branching
else:
    #choose an index
    if variable_search == 0:
        #choose the index which the variable is furthest away from integer
        i_search = np.ma.argmin(abs(integer_test-0.5))
    elif variable_search == 1:
        #choose the first minimal index
        i_search = np.ma.argmin(integer_test)
    elif variable_search == 2:
        #choose the first maximal index
        i_search = np.ma.argmax(integer_test)
    elif variable_search == 3:
        #choose the first possible index
        i_search = min(np.ma.flatnotmasked_edges(integer_test))
    elif variable_search == 4:
        #choose with strong branching
        if integer_test.count() == 1:
            i_search = min(np.ma.flatnotmasked_edges(integer_test))
        else:
            i_search = strong_branching(pos[3][k], integer_test, c)



    #define new nodes
    max_index = max(pos[0])

    l = i_search
    input_v_1 = np.eye(len(c),1,1-(l+1))
    input_v_1.shape = (len(c))
    input_b_1 = np.floor(pos[3][k]['x_opt'][i_search])
    Node_1 = dsimplex(pos[3][k], input_v_1, input_b_1, maxiter=1000)
    if Node_1['bound'] == 0:
        if Node_1['f_opt'] <= f_opt:
```

```
            add_1 = np.array([[(max_index)+1], [0], [0], [0], [np.nan]])
        else:
            add_1 = np.array([[(max_index)+1], [pos[1][k] + 1], [0], [Node_1], \
                    [Node_1['f_opt']]])
            index.add((max_index)+1)
    #if dsimplex maxiter reached define new variable search
    elif Node_1['bound'] == 3:
        variable_search = np.mod(variable_search+1, 3)
    else:
        add_1 = np.array([[(max_index)+1], [0], [0], [0], [np.nan]])

    input_v_2 = -1*input_v_1
    input_b_2 = -1*(input_b_1 + 1)
    Node_2 = dsimplex(pos[3][k], input_v_2, input_b_2, maxiter=1000)
    if Node_2['bound'] == 0:
        if Node_2['f_opt'] <= f_opt:
            add_2 = np.array([[(max_index)+2], [0], [1], [0], [np.nan]])
        else:
            add_2 = np.array([[(max_index)+2], [pos[1][k] + 1], [1], [Node_2], \
                    [Node_2['f_opt']]])
            index.add((max_index)+2)
    elif Node_2['bound'] == 3:
        variable_search = np.mod(variable_search+1, 3)
    else:
        add_2 = np.array([[(max_index)+2], [0], [1], [0], [np.nan]])
    pos = np.hstack([pos,add_1,add_2])
    index.discard(k)
    pos[1][k] = 0
    pos[3][k] = 0
    pos[4][k] = np.nan

count = int(np.max(pos[0])) + P_initial['iterations']
comptime = (time.time() - start) + P_initial['time']

if bound == 3:
    if disp == 1:
        print("maxiter is reached")
        print("-------------------------------------------")
        print("number of iterations:" , count)
        print("-------------------------------------------")
        print("time:" , comptime)
        print("-------------------------------------------")
        print("try it again with another options")
        print("________***end of program***_________________")
    return {'time': comptime, 'iterations': count, 'f_opt': f_opt, 'x_opt': x_opt, \
            'bound': bound}

elif type(x_opt) == int:
    bound = 4
    if disp == 1:
        print("the problem has no integer solution")
        print("-------------------------------------------")
        print("number of iterations:" , count)
        print("-------------------------------------------")
        print("time:" , comptime)
        print("-------------------------------------------")
        print("________***end of program***_________________")
    return {'time': comptime, 'iterations': count, 'f_opt': f_opt, 'x_opt': x_opt, \
            'bound': bound}

else:
```

```
if disp == 1:
    print("optimal value of the costfunction:")
    print(f_opt)
    print("----------------------------------------------")
    print("number of iterations:" , count)
    print("----------------------------------------------")
    print("time:" , comptime)
    print("----------------------------------------------")
    print("_______***end of program***_________________")
return {'time': comptime, 'iterations': count, 'f_opt': f_opt, 'x_opt': x_opt, \
        'bound': bound}
```

## 6.4 Main File Branch-and-Bound

### 6.4.1 Python

```
from IPython import get_ipython
get_ipython().magic('reset -sf')



import numpy as np
from branch_and_bound import b_and_b


import random
random.seed()

"""
c = np.array([17,12])
N = np.array([[10, 7],[1,1]])
b = np.array([40,5])
P = b_and_b(c ,N ,b, disp=1, variable_search = 4, initial = 'deepfirst', nodesearch = 'deepfirst')

#model Kämi
c = np.array([1000, 700])
N = np.array([[100,50],[0,20]])
b = np.array([2425,510])
P = b_and_b(c, N, b, disp=1, variable_search = 3, initial = 'deepfirst', nodesearch = 'deepfirst')


#model Rucksack
c = np.array([5,3,6,6,2])
N = np.array([[5,4,7,6,2],[1,0,0,0,0],[0,1,0,0,0],[0,0,1,0,0],[0,0,0,1,0],[0,0,0,0,1]])
b = np.array([15,1,1,1,1,1])
P = b_and_b(c,N,b,disp=1, variable_search=4, initial = 'bestsolution', nodesearch = 'bestsolution')

"""
import matlab.engine
eng = matlab.engine.start_matlab()



#_____________________________________________________________________________


Zeitb_and_b = []
ZeitMatlab = []
Iterb_and_b = []
IterMatlab = []
Vgl_f_opt_b_and_b = []
Vgl_f_opt_matlab = []
Wert_matlab = []
```

```
Wert_b_and_b = []


primal = 0
dual = 0
nothing = 0

unbound = 0
empty = 0
noint = 0
allright = 0
maxiter_reached_1 = 0
maxiter_reached_2 = 0



#random testing
m = 30          #choose the dimension of the unknown
n = 36          #choose the dimension of the contraints
integer = 10     #choose how big the integers



k = 0
while k < 100:

    #define random number to choose the problem
    randomnumber = random.randint(0,2)

    #primal
    if randomnumber == 0:
        #define a primal feasible problem
        primal = primal + 1
        c = np.zeros(m)
        for i in range(0,m):
            c[i] = random.randint(-integer,integer)



        b = np.zeros(n)
        for i in range(1,n):
            b[i] = random.randint(0,10*integer)


        N = np.zeros((n,m))
        for i in range(0,n):
            for j in range(0,m):
                N[i][j] = random.randint(-integer,integer)

    #dual
    if randomnumber == 1:
        #define a dual feasible problem
        dual = dual + 1
        c = np.zeros(m)
        for i in range(0,m):
            c[i] = random.randint(-integer,0)



        b = np.zeros(n)
        for i in range(1,n):
```

```
        b[i] = random.randint(-integer,10*integer)


    N = np.zeros((n,m))
    for i in range(0,n):
        for j in range(0,m):
            N[i][j] = random.randint(-integer,integer)



#both
if randomnumber == 2:
    #define a not primal feasible and not dual feasible problem
    nothing = nothing +1
    c = np.zeros(m)
    for i in range(0,m):
        c[i] = random.randint(-integer,integer)



    b = np.zeros(n)
    for i in range(1,n):
        b[i] = random.randint(-integer,10*integer)


    N = np.zeros((n,m))
    for i in range(0,n):
        for j in range(0,m):
            N[i][j] = random.randint(-integer,integer)



#save the linear program to compare it with matlab
np.savetxt("c_general.txt", c, fmt="%2.3f", delimiter=",")
np.savetxt("b_general.txt", b, fmt="%2.3f", delimiter=",")
np.savetxt("N_general.txt", N, fmt="%2.3f", delimiter=",")



P = b_and_b(c, N, b, disp = 0, variable_search = 5, initial = 'deepfirst', nodesearch =
Wert_b_and_b.append(P['f_opt'])
#solve with matlab
[f_opt_matlab, Iter_matlab, exitflag, time_matlab] = eng.comparison_matlab_python_b_and_b(nargout =
Wert_matlab.append(f_opt_matlab)



#save time, iter, f_opt to compare the values
if P['bound'] == 0 and exitflag == 1:
    allright += 1
    Vgl_f_opt_b_and_b.append(P['f_opt'])
    Vgl_f_opt_matlab.append(f_opt_matlab)

Zeitb_and_b.append(P['time'])
ZeitMatlab.append(time_matlab)
Iterb_and_b.append(P['iterations'])
IterMatlab.append(Iter_matlab)
```

```
    #check if solution found
    if P['bound']==1 and exitflag == -3:
        #
        unbound += 1

    if P['bound']==2 and exitflag == -2:
        #
        empty += 1

    if P['bound'] == 4 and exitflag == 0:
         #
        noint += 1

    if P['bound'] == 3:
        maxiter_reached_1 += 1

    if Iter_matlab >= 10**(6):
        maxiter_reached_2 += 1



    k += 1
    #if k == 10 or k == 20 or k == 30 or k == 50 or k == 70 or k == 90:
    print(k)


print("-------------------------------------------------------------------------")
print("arithmetic mean of time by using b_and_b:" , np.mean(Zeitb_and_b))
print("arithmetic mean of time by using matlab: " ,  np.mean(ZeitMatlab))
print("-------------------------------------------------------------------------")
print("arithmetic mean of iterations by using b_and_b:" , np.mean(Iterb_and_b))
print("arithmetic mean of iterations by using matlab: " , np.mean(IterMatlab))
print("-------------------------------------------------------------------------")
print("arithmetic mean of f_opt by using b_and_b", np.mean(Vgl_f_opt_b_and_b))
print("arithmetic mean of f_opt by using matlab ", np.mean(Vgl_f_opt_matlab))
print("-------------------------------------------------------------------------")
print("number of problems which both found an opt          ", allright)
print("number of problems which the problems are unbound    ", unbound)
print("number of problems which the allowed space are empty", empty)
print("number of problems without an integer soluten       ", noint)
print("number of problems where maxiter is reached         ", maxiter_reached_1)
print("number of problems where maxiter is reached         ", maxiter_reached_2)
print("-------------------------------------------------------------------------")
```

### 6.4.2 Matlab

```
function [f_opt, iter, exitflag, time] = comparison_matlab_python_b_and_b()

% c : vector with coefficients of the linear objective function; because of
% Matlab minimize the Problem, we multply it with -1
c = -1*load('c_general.txt');
%because the whole vector has to be integer, set intcon equal to the vector
%from 1 to lenght(c)
intcon = [1:length(c)];
% N : matrix with coefficients of the linear inequality constraints
N = load('N_general.txt');
% bineq : vector for linear inequality constraints
b = load('b_general.txt');
% lb : lower bound constraints (no upper bound constraints -> set ub =[]
lb = zeros (length(c),1) ;
options = optimoptions ( @intlinprog, 'BranchRule', 'maxfun', 'CutGeneration',...
    'none', 'Heuristics', 'none', 'IntegerPreprocess', 'none',...
    'IntegerTolerance', 1e-6, 'LPPreprocess', 'none', 'NodeSelection', 'minobj') ;
% x: solution x
% fval : value of the objective function at the solution x
% exitflag : informs about the reason for termination
% output : Gives information about the optimization process e . g . the number
% of iterations , the used algorithm or an exit message .
tic;
[x , fval , exitflag , output] = intlinprog(c, intcon, N, b , [ ] , [ ] ,                    ...
    lb, [ ], options);
toc

%output
f_opt = -fval;
iter = output.numnodes;
%exitflag = exitflag;
time = toc;
```