

Lineare und Mixed-Integer Programmierung

■ Eine allererste Einführung

Dozent: Prof. Dr. Michael Eichberg

Kontakt: michael.eichberg@dhbw.de, Raum 149B

Version: 1.0.4

Quelle: Introduction to Algorithms, 3rd Edition, Cormen, Leiserson, Rivest, Stein, MIT Press, 2009



1

Folien: https://delors.github.io/theo-algo-mixed_integer_programming/folien.de.rst.html
https://delors.github.io/theo-algo-mixed_integer_programming/folien.de.rst.html.pdf

Fehler melden:
<https://github.com/Delors/delors.github.io/issues>

1. Einführung in lineare Programmierung

Beispielszenario: Kostenoptimierung

■ Optimierung der Kosten für die Nahrungsmittelzusammensetzung

Seien x_1 und x_2 die Menge an Nahrungsmitteln 1 und 2, die wir kaufen. Die Kosten für Nahrungsmittel 1 und 2 betragen 1 und 2 Euro pro Einheit. Die täglichen Ernährungsbedürfnisse sind 10 Einheiten Protein und 20 Einheiten Fett. Nahrungsmittel 1 enthält 2 Einheiten Protein und 3 Einheiten Fett pro Einheit. Nahrungsmittel 2 enthält 1 Einheit Protein und 4 Einheiten Fett pro Einheit.

Zielfunktion (🚩 *objective function* oder einfach nur 🚩 *objective*)

$$\text{minimiere } x_1 \cdot 1\text{€} + x_2 \cdot 2\text{€}$$

(unter den) Nebenbedingungen (🚩 *constraints*/🚩 *subject to (s.t.)*)

$$\begin{aligned} 2 \cdot x_1 + 1 \cdot x_2 &\geq 10 && \text{Nebenbedingung bzgl. Protein} \\ 3 \cdot x_1 + 4 \cdot x_2 &\geq 20 && \text{Nebenbedingung bzgl. Fett} \\ x_1, x_2 &\geq 0 && \end{aligned}$$

Lineare Programmierung

Definition

Lineare Programmierung: Optimierung von linearen Funktionen unter linearen Nebenbedingungen.

Das Ziel ist die Optimierung (Maximierung/Minimierung) einer linearen Funktion f :

$$f(x_1, \dots, x_n) = a_1 \cdot x_1 + a_2 \cdot x_2 + \dots + a_n \cdot x_n = \sum_{i=1}^n a_i \cdot x_i$$

Unter einer Menge von linearen Nebenbedingungen. Sei $b \in \mathbb{R}$, dann ist ...

- eine *lineare Ungleichung* der Form: $f(x_1, \dots, x_n) \leq b$
- eine *lineare Gleichung* der Form: $f(x_1, \dots, x_n) = b$
- lineare Ungleichungen und Gleichungen beschreiben die *linearen Nebenbedingungen*.

Lösen von linearen Optimierungsproblemen

Standardform - „nur“ Verwendung von linearen Ungleichungen

Zielfunktion (Maximiere)

$$x_1 + x_2$$

Nebenbedingungen

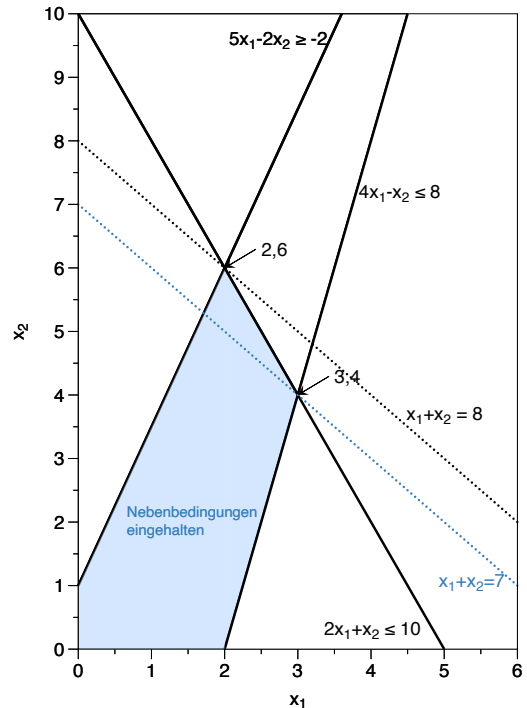
$$4x_1 - x_2 \leq 8$$

$$2x_1 + x_2 \leq 10$$

$$5x_1 - 2x_2 \geq -2$$

$$x_1 \geq 0$$

$$x_2 \geq 0$$



Schlupfform (🚧 *Slack Form*) - „nur“ Verwendung von linearen Gleichungen

Zielfunktion (Maximiere)

$$x_1 + x_2$$

unter den Nebenbedingungen

$$\begin{aligned}
 x_3 &= 8 - 4x_1 + x_2 \\
 x_4 &= 10 - 2x_1 - x_2 \\
 x_5 &= 2 + 5x_1 - 2x_2 \\
 0 &\leq x_1, x_2, x_3, x_4, x_5
 \end{aligned}$$

Die Variablen x_3 , x_4 und x_5 sind die Schlupfvariablen. Sie messen die Differenz zwischen der linken und der rechten Seite der Ungleichungen und sind nicht Teil der Zielfunktion.

Beobachtungen (am Beispiel orientiert)

- der Bereich der zulässigen Lösungen enthält (im Allgemeinen) unendlich viele Punkte
- der Bereich der zulässigen Lösungen ist beschränkt/ist (hier) ein konvexes Polygon (im Allgemeinen ein konvexes Polyeder)
- Die konvexe Hülle einer endlichen Anzahl von affin unabhängigen Punkten in einem n-dimensionalen Raum bezeichnen wir als Simplex
- in diesem (2-Dimensionalen) Fall können wir die Lösung grafisch darstellen
- nicht jedes lineare Optimierungsproblem hat eine (bzw. eine optimale) Lösung
- Auch in der Schlupfform, werden die Anforderungen an die nicht-Negativität der Variablen als Ungleichungen beschrieben.

Affine Unabhängigkeit:

Zwei Punkte sind affin unabhängig, wenn die Differenz der beiden Punkte nicht durch einen Skalarfaktor auf den anderen Punkt abgebildet werden kann. (Im 2-D Raum: Die beiden Punkte liegen nicht auf einer Geraden, wenn die beiden Punkte als entsprechende Vektoren aufgefasst werden.)

Die Schlupfform ist für den Simplex-Algorithmus relevant.

■ Formulierung eines linearen Programms

In einem Betrieb mit mehrschichtiger Arbeitszeit besteht folgender Mindestbedarf an Personal:

von 0 bis 4 Uhr: 3 Personen

von 4 bis 8 Uhr: 8 Personen

von 8 bis 12 Uhr: 10 Personen

von 12 bis 16 Uhr: 8 Personen

von 16 bis 20 Uhr: 14 Personen

von 20 bis 24 Uhr: 5 Personen

Der Arbeitsbeginn ist jeweils um 0, 4, 8, 12, 16 bzw. 20 Uhr. Die Arbeitszeit beträgt stets 8 Stunden hintereinander. Formulieren Sie ein lineares Program, um einen Einsatzplan mit minimalem Gesamtpersonalbedarf aufzustellen.

[1] Aus: Übungsbuch Operations Research; Domschke, Drexl, Schildt, Scholl, Voß; Springer Verlag 1997

Formulierung eines linearen Programms

In einem Betrieb mit mehrschichtiger Arbeitszeit besteht folgender Mindestbedarf an Personal:

von 0 bis 4 Uhr: 3 Personen

von 4 bis 8 Uhr: 8 Personen

von 8 bis 12 Uhr: 10 Personen

von 12 bis 16 Uhr: 8 Personen

von 16 bis 20 Uhr: 14 Personen

von 20 bis 24 Uhr: 5 Personen

Der Arbeitsbeginn ist jeweils um 0, 4, 8, 12, 16 bzw. 20 Uhr. Die Arbeitszeit beträgt stets 8 Stunden hintereinander. Formulieren Sie ein lineares Program, um einen Einsatzplan mit minimalem Gesamtpersonalbedarf aufzustellen.

Berechnung des maximalen Flusses (Maximum-Flow-Problem)

Formulieren Sie ein lineares Programm zur Bestimmung des maximalen Flusses von einer Quelle s zu einer Senke t in einem Netzwerk mit V Knoten. (Die Funktion) f_{uv} sei der Fluss zwischen zwei Knoten u und v . Nebenbedingungen:

Kapazitätsbeschränkung:

Der Fluss f_{uv} auf einer Kante darf die Kapazität ($c(u, v)$) der Kante nicht überschreiten.

Flusserhaltung:

Für jeden Knoten (außer Quelle und Senke) gilt, dass der zufließende Fluss gleich dem abfließenden Fluss ist.

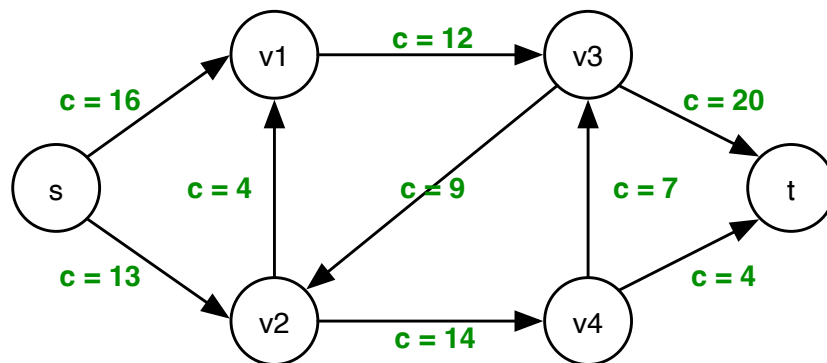
Richtungsabhängigkeit:

Der Fluss ist gerichtet (von einem Knoten zum anderen).

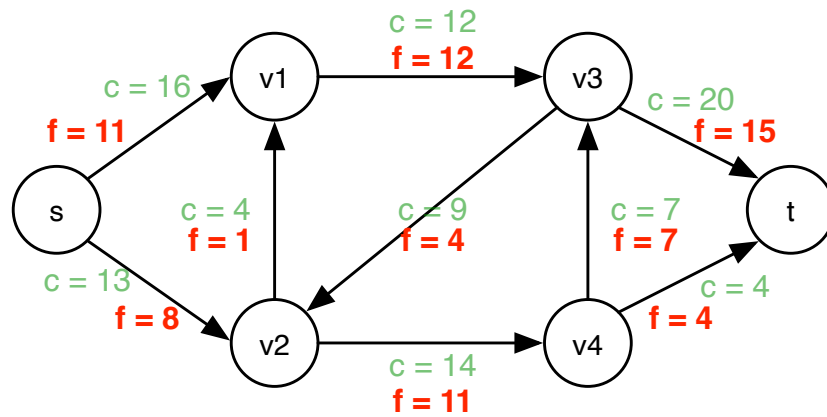
Sie können die vereinfachende Annahme machen, dass die Summe der Zuflüsse zur Quelle 0 ist ($\sum_{v \in V} f_{vs} = 0$); dass die Quelle keine eingehenden Kanten hat. Weiterhin sei die Kapazität zwischen zwei nicht-verbundenen Knoten 0 .

Beispiel

Netzwerk mit Kapazitäten:



Eine (nicht notwendigerweise optimale) Lösung, die die Nebenbedingungen erfüllt:



Im Allgemeinen gilt

Das Netzwerk ist modelliert als gerichteter Graph $G = (V, E)$ ohne Eigenschleifen und ohne antiparallele Kanten (d. h. $(v, u) \in E \Rightarrow (u, v) \notin E$). Jeder Kante $(u, v) \in E$ ist eine nicht-negative Kapazität $c(u, v) \geq 0$ zugeordnet.

Sei $(u, v) \notin E$, dann ist $c(u, v) = 0$.

Empfohlene Vorgehensweise

1. Bestimmen Sie die Zielfunktion in Hinblick auf den Fluss bzw. der Variablen, die den Fluss repräsentieren.
2. Formulieren Sie die Nebenbedingungen:
 1. in Hinblick darauf, dass der Fluss über eine Kante nie negativ sein darf
 2. in Bezug auf die Kanten und die Kapazitäten
 3. in Bezug auf die Kapazitätserhaltung

Berechnung des maximalen Fluss (Maximum-Flow-Problem)

Formulieren Sie ein lineares Programm zur Bestimmung des maximalen Flusses von einer Quelle s zu einer Senke t in einem Netzwerk mit V Knoten. (Die Funktion) f_{uv} sei der Fluss zwischen zwei Knoten u und v . Nebenbedingungen:

Kapazitätsbeschränkung:

Der Fluss f_{uv} auf einer Kante darf die Kapazität ($c(u, v)$) der Kante nicht überschreiten.

Flusserhaltung:

Für jeden Knoten (außer Quelle und Senke) gilt, dass der zufließende Fluss gleich dem abfließenden Fluss ist.

Richtungsabhängigkeit:


Der Fluss ist gerichtet (von einem Knoten zum anderen).

Sie können die vereinfachende Annahme machen, dass die Summe der Zuflüsse zur Quelle 0 ist ($\sum_{v \in V} f_{vs} = 0$); dass die Quelle keine eingehenden Kanten hat. Weiterhin sei die Kapazität zwischen zwei nicht-verbundenen Knoten 0.

2. Simplex Algorithmus

Simplex Algorithmus - Einführung

- Der Simplex-Algorithmus ist ein Algorithmus zur Lösung von linearen Optimierungsproblemen.
- Der Algorithmus wurde von George Dantzig entwickelt und 1947 veröffentlicht.
- Der Simplex-Algorithmus ist ein iteratives Verfahren, das in der Regel sehr effizient ist
Im Regelfall polynomielle Laufzeit, im Worst-case jedoch exponentiell.
- Der Simplex-Algorithmus ist ein Beispiel für einen Algorithmus, der auf einem Netzwerk von Kanten operiert.

Der Simplex-Algorithmus bewegt sich systematisch entlang der Ecken ( *Vertices*) des Bereichs, der die zulässigen Lösungen des linearen Programms beschreibt, um die optimale Lösung zu finden. Er terminiert, wenn er das lokale Optimum erreicht hat. Aufgrund der konvexen Natur des Problems ist das lokale Optimum gleichzeitig das globale Optimum.

Es gibt Algorithmen, die eine garantierte polynomielle Laufzeit haben, wie zum Beispiel der Ellipsoid-Algorithmus. Der Simplex-Algorithmus ist jedoch in der Praxis oft schneller.

Standardform

Gegeben sein n reelle Zahlen (c_1, \dots, c_n) ; m reelle Zahlen (b_1, \dots, b_m) ; und eine $m \times n$ Matrix $A = (a_{ij})$ für $i = 1, 2, \dots, m$ und $j = 1, 2, \dots, n$.

Wir möchten nun n reelle Zahlen (x_1, \dots, x_n) finden, die die folgenden Bedingungen erfüllen:

Zielfunktion (📌 *objective function*)

$$\text{maximiere } \sum_{j=1}^n c_j \cdot x_j$$

(unter den) Nebenbedingungen (📌 *subject to/constraints*)

$$\begin{aligned} \sum_{j=1}^n a_{ij} \cdot x_j &\leq b_i \quad \text{für } i = 1, 2, \dots, m \\ x_j &\geq 0 \quad \text{für } j = 1, 2, \dots, n \end{aligned}$$

Kompakte Darstellung

Gegeben Matrix $A = (a_{ij})$, m -Vektor $b = (b_i)$, n -Vektor $c = (c_j)$, und n -Vektor $x = (x_j)$.

Dann ist das lineare Programm:

$$\begin{aligned} &\text{maximiere } c^T x \\ &\text{unter den Nebenbedingungen } A \cdot x \leq b \\ & \quad \quad \quad x \geq 0 \end{aligned}$$

Terminologie

zulässige Lösung/📌 *feasible*:

Eine Belegung der Variablen \bar{x} , die die Nebenbedingungen erfüllt.

optimale Lösung:

Eine zulässige Lösung, die die Zielfunktion maximiert.

unbeschränkt:

Ein lineares Programm, das Lösungen hat, die die Zielfunktion nicht beschränken.

unzulässig/📌 *infeasible*:

Ein lineares Programm, das keine zulässige Lösung hat.

Konvertierung von beliebigen linearen Programmen in die Standardform

Ein lineares Programm kann aus folgenden vier Gründen nicht in Standardform sein:

- Die Zielfunktion ist zu minimieren
- Es gibt Variablen ohne Nichtnegativitätsbedingung
- Es gibt Gleichungen (=)
- Es gibt Ungleichungen mit \geq statt \leq

Regeln

1. Minimierungsprobleme können durch Multiplikation der Zielfunktion mit -1 in ein Maximierungsproblem umgewandelt werden.
2. Variablen ohne Nichtnegativitätsbedingung können durch die Einführung von Differenzvariablen in Nichtnegativitätsbedingungen umgewandelt werden.

D. h. wir ersetzen die Vorkommen der Variablen $c \cdot x$ durch $c \cdot x^+ - c \cdot x^-$ wobei x^+ und x^- nicht-negativ sind.
3. Gleichungen können in zwei Ungleichungen umgewandelt werden.
4. Ungleichungen mit \geq können durch Multiplikation mit -1 in Ungleichungen mit \leq umgewandelt werden.

-
- $c^T x$ ist das innere Produkt.
 - $x \geq 0$ bedeutet, dass jede Komponente von x nicht negativ sein darf.

Schlupfform (🇺🇸 *Slack Form*)

- Zum effizienten Lösung von linearen Programmen wird die Schlupfform verwendet.
- Bei der Schlupfform werden alle Nebenbedingungen in Gleichungen umgewandelt - abgesehen von den Nichtnegativitätsbedingungen.

Vorgehen

Gegeben sei *eine* Ungleichung:

$$\sum_{j=1}^n a_{ij} \cdot x_j \leq b_i \quad (1)$$

Wir führen dann eine Schlupfvariable (🇺🇸 *slack variable*) x_{n+i} ein und erhalten:

$$x_{n+1} = b_i - \sum_{j=1}^n a_{ij} \cdot x_j \quad (2)$$

$$x_{n+1} \geq 0 \quad (3)$$

- Somit stehen die Variablen x_1, \dots, x_n für die ursprünglichen Variablen und die Variablen x_{n+1}, \dots, x_{n+m} für die Schlupfvariablen.
- Auf der rechten Seite der Gleichung (2) stehen die ursprünglichen Variablen. Nur diese Variablen sind (initial) Teil der Zielfunktion.

Basisvariablen:

Die Variablen auf der linken Seite der Gleichung (2).

Nichtbasisvariablen:

Die Variablen auf der rechten Seite der Gleichung (2).

- Für den Wert der Zielfunktion verwenden wir die Variable z .
- Im Folgenden gilt:

N : die Menge der Indizes der Nichtbasisvariablen

B : die Menge der Indizes der Basisvariablen

v : ein optionaler konstanter Term in der Zielfunktion

$$|N| = n, |B| = m \text{ und } N \cup B = 1, \dots, n + m$$

Somit ist die kompakte Darstellung des linearen Programms in Schlupfform:

$$\begin{aligned} z &= v + \sum_{j \in N} c_j \cdot x_j \\ x_i &= b_i - \sum_{j \in N} a_{ij} \cdot x_j \quad \text{für } i \in B \end{aligned}$$

Diese Schlupfvariable (x_{n+1}) misst die Differenz zwischen der linken und der rechten Seite der Ungleichung (1).

Überführen eines linearen Programms in Schlupfform

Überführen Sie das 1. lineare Programm aus der vorhergehenden Übung in Schlupfform.

Bauen Sie ggf. auf den Ergebnissen der vorhergehenden Aufgabe auf.

Zeigen Sie (grafisch), dass das folgende lineare Programm unbeschränkt ist.

$$\begin{array}{rllll} & \text{maximiere} & x_1 & - & x_2 \\ \text{unter den Nebenbedingungen} & & -2x_1 & + & x_2 & \leq & -1 \\ & & -x_1 & - & 2x_2 & \leq & -2 \\ & & x_1, & x_2 & & \geq & 0 \end{array}$$

(Primaler) Simplex

Grundlegende Idee

Wir lösen unser Optimierungsproblem durch gezielte algebraische Operationen, die die Zielfunktion maximieren.

1. Wir wählen immer eine Variable, die in der Zielfunktion vorkommt und einen positiven Koeffizienten hat.
(D. h. wir wählen eine Variable deren Erhöhung die Zielfunktion maximiert.)
2. Dann bestimmen wir die Ungleichung, die die Maximierung der gewählten Variable am stärksten einschränkt.
3. Wir „tauschen“ die Variable mit der Schlupfvariablen, die in dieser Ungleichung vorkommt und lösen die Gleichung nach der gewählten Variable auf.
4. Wir setzen dann die umgestellte Gleichung in alle anderen Gleichungen (inkl. Zielfunktion) ein, um die Werte der anderen Variablen zu bestimmen.

Wir nennen diesen Prozess (1-4) „Pivot Operation“.

Wir wiederholen diesen Prozess, bis wir keine Variable mehr finden, die die Zielfunktion maximiert. An dieser Stelle können wir dann das Optimum und die Werte für die Variablen (x_1, \dots, x_{n+m}) ablesen.

Es ist in Hinblick auf die Korrektheit gleichgültig welche Variable wir im ersten Schritt wählen. Es kommt jedoch ggf. zu einer unterschiedlichen Anzahl an Schritten, bis wir die optimale Lösung finden.

Simplex anwenden

Gegebenes lineares Programm in Schlupfform

Wir führen die Schlupfvariablen x_4 , x_5 und x_6 ein mit der Nebenbedingung: $x_4, x_5, x_6 \geq 0$

$$\begin{array}{rcll} \text{maximiere} & z & = & 3x_1 + x_2 + 2x_3 \\ \text{unter den Nebenbedingungen} & x_4 & = & 30 - x_1 - x_2 - 3x_3 \\ & x_5 & = & 24 - 2x_1 - 2x_2 - 5x_3 \\ & x_6 & = & 36 - 4x_1 - x_2 - 2x_3 \end{array}$$

- Wir können die Zielfunktion maximieren, indem wir die Variable der Zielfunktion mit dem größten positiven Koeffizienten wählen: x_1 .
- Wir prüfen welche Nebenbed. die Maximierung von x_1 am stärksten einschränkt:
1. Nebenbed.: $x_1 \leq 30$, 2. Nebenbed.: $x_1 \leq 12$ und 3. Nebenbed.: $x_1 \leq 9$
- Die (nicht-Basis)Variable x_1 wird somit durch die Schlupfvariable/Basisvariable x_6 ersetzt:

$$4x_1 = 36 - x_6 - x_2 - 2x_3 \Rightarrow x_1 = 9 - \frac{1}{4}x_6 - \frac{1}{4}x_2 - \frac{1}{2}x_3$$

- Wir setzen x_1 in die Zielfunktion und die anderen Nebenbedingungen ein und erhalten:

$$x_4 = 30 - \left(9 - \frac{1}{4}x_6 - \frac{1}{4}x_2 - \frac{1}{2}x_3\right) - x_2 - 3x_3$$

$$x_4 = 21 + \frac{1}{4}x_6 - \frac{3}{4}x_2 - \frac{5}{2}x_3$$

- Ergebnis

$$\begin{array}{rcll} z & = & 27 & - \frac{3}{4}x_6 + \frac{1}{4}x_2 + \frac{1}{2}x_3 \\ x_1 & = & 9 & - \frac{1}{4}x_6 - \frac{1}{4}x_2 - \frac{1}{2}x_3 \\ x_4 & = & 21 & + \frac{1}{4}x_6 - \frac{3}{4}x_2 - \frac{5}{2}x_3 \\ x_5 & = & 6 & - \frac{1}{2}x_6 - \frac{3}{2}x_2 - 4x_3 \end{array}$$

Diese Operation wird als Pivot Operation bezeichnet.

- Im nächsten Schritt könnten wir x_3 wählen, da es den größten positiven Koeffizienten hat. Da die dritte Nebenbedingung die Maximierung von x_3 am stärksten einschränkt, würden wir x_3 durch die Schlupfvariable x_5 ersetzen.
- Ergebnis

$$\begin{aligned}
z &= \frac{111}{4} + \frac{1}{16}x_2 - \frac{1}{8}x_5 - \frac{11}{16}x_6 \\
x_1 &= \frac{33}{4} - \frac{1}{16}x_2 + \frac{1}{8}x_5 - \frac{5}{16}x_6 \\
x_3 &= \frac{3}{2} - \frac{3}{8}x_2 - \frac{1}{4}x_5 + \frac{1}{8}x_6 \\
x_4 &= \frac{69}{4} + \frac{3}{16}x_2 + \frac{5}{8}x_5 - \frac{1}{16}x_6
\end{aligned}$$

- Die Basislösung ist: $(33/4, 0, 3/2, 69/4, 0, 0)$ und der Wert der Zielfunktion ist $111/4$.
- Im letzten Schritte würden wir x_2 wählen. Da die zweite Nebenbedingung die Maximierung von x_2 am stärksten einschränkt, würden wir x_2 durch die Schlupfvariable x_3 ersetzen.
- Ergebnis

$$\begin{aligned}
z &= 28 - \frac{1}{6}x_3 - \frac{1}{6}x_5 - \frac{2}{3}x_6 \\
x_1 &= 8 + \frac{1}{6}x_3 + \frac{1}{6}x_5 - \frac{1}{3}x_6 \\
x_2 &= 4 - \frac{8}{3}x_3 - \frac{2}{3}x_5 + \frac{1}{3}x_6 \\
x_4 &= 18 + \frac{1}{2}x_3 + \frac{1}{2}x_5
\end{aligned}$$

- Die Basislösung ist: $(8, 4, 0, 18, 0, 0)$ und der Wert der Zielfunktion ist 28 .
- Eine weitere Verbesserung der Zielfunktion ist nicht möglich. Die Basislösung ist somit unsere optimale Lösung.

Beobachtungen:

- Beim Start: jede Belegung der Variablen x_1, \dots, x_3 definiert Werte für die Variablen x_4, \dots, x_6 und ist somit eine Lösung.
- eine Lösung ist (jedoch nur) dann zulässig wenn alle Variablen nicht-negativ sind.
- Die Basislösung ist die Lösung, bei der die nicht-Basisvariablen (im ersten Schritt also x_1, x_2 und x_3) den Wert 0 haben. Im ersten Schritt ergibt sich somit die Basislösung $(\bar{x}_1, \dots, \bar{x}_6) (0, 0, 0, 30, 24, 36)$; der Wert der Zielfunktion ist 0 .

Simplex Algorithmus

```
1 Algorithm Simplex(A,b,c):
2   (N,B,A,b,c,v) := InitialisiereSimplex(A,b,c)
3   sei  $\Delta$  ein Vektor der Länge m
4   while  $\exists j \in N$  mit  $c_j > 0$  do
5     wähle Index  $e \in N$  mit  $c_e > 0$  { e für "entering variable" }
6     for Index  $i \in B$ 
7        $\Delta_i := b_i / A_{ie}$  falls  $A_{ie} > 0$ , sonst  $\infty$ 
8     wähle  $l \in B$  mit  $\Delta_l := \min(\Delta_1, \dots, \Delta_m)$ 
9     if  $\Delta_l = \infty$  then return "unbeschränkt"
10    (N,B,A,b,c,v) := Pivot(N,B,A,b,c,v,l,e)
11  for  $i := 1$  to n { Gib die Lösung zurück }
12    if  $i \in B$  then
13       $x_i := b_i$ 
14    else
15       $x_i := 0$ 
16  return  $(x_1, \dots, x_n)$ 
```

17

InitialisiereSimplex(A,b,c)

Falls das LP lösbar ist, dann gib das LP in Schlupfform zurück, in der die initiale Basislösung zulässig ist.

Wir werden uns im Rahmen dieses Kurses nicht weiter mit der Implementierung des Simplex-Algorithmus beschäftigen. Es ist jedoch wichtig, dass Sie die Funktionsweise des Algorithmus verstehen.

3. Mixed-Integer- Programmierung (MIP)

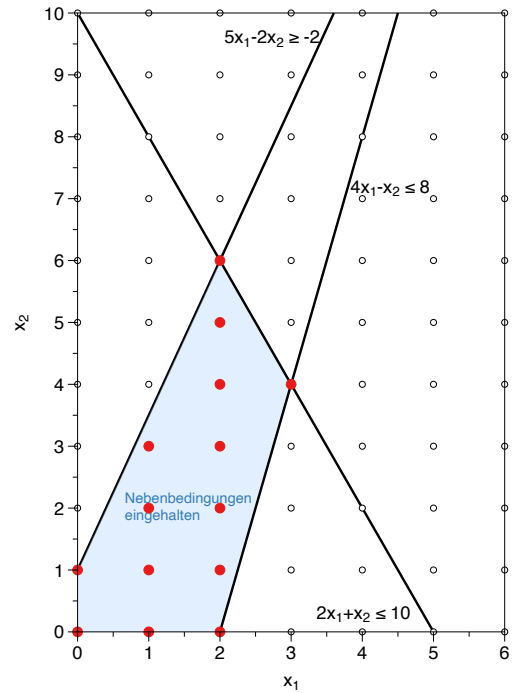
MIP: einige (oder alle) Variablen sind ganzzahlig

Zielfunktion (Maximiere)

$$x_1 + x_2$$

Nebenbedingungen

$$\begin{aligned} 4x_1 - x_2 &\leq 8 \\ 2x_1 + x_2 &\leq 10 \\ 5x_1 - 2x_2 &\geq -2 \\ x_1 &\geq 0 \text{ und ganzzahlig} \\ x_2 &\geq 0 \text{ und ganzzahlig} \end{aligned}$$

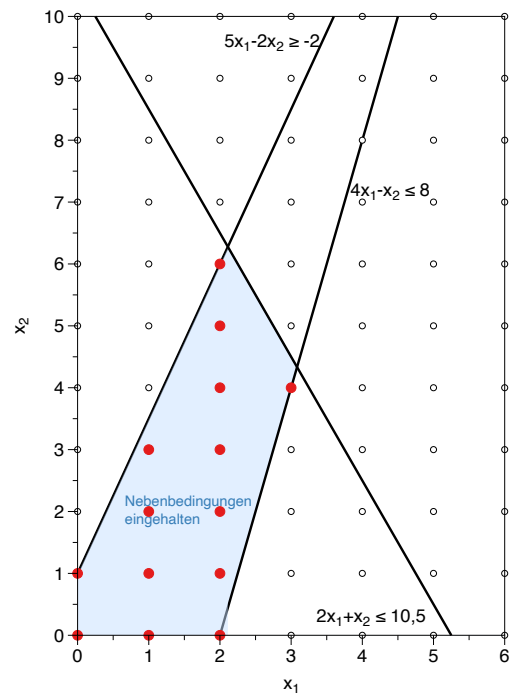


Zielfunktion (Maximiere)

$$x_1 + x_2$$

Nebenbedingungen

$$\begin{aligned} 4x_1 - x_2 &\leq 8 \\ 2x_1 + x_2 &\leq 10,5 \\ 5x_1 - 2x_2 &\geq -2 \\ x_1 &\geq 0 \text{ und ganzzahlig} \\ x_2 &\geq 0 \text{ und ganzzahlig} \end{aligned}$$



Hinweis

Durch die Einschränkung, dass die Variablen ganzzahlig sein müssen, wird das Problem schwieriger zu lösen und ist NP-schwer, während das lineare Programm in polynomieller Zeit gelöst werden kann.

Zur Lösung von MIPs gibt es verschiedene Ansätze, wie z. B. den Branch-and-Bound-Algorithmus, bzw. Branch-and-Cut-Algorithmus. Häufig werden in der Praxis auch

20

Wenn alle Variablen ganzzahlig sind, sprechen wir von einem reinen ganzzahligen Programm (🇺🇸 *Integer Programming*).
Wenn nur einige Variablen ganzzahlig sind, sprechen wir von einem gemischt ganzzahligem Programm.

Wir konzentrieren uns im Folgenden darauf für konkrete Probleme, ganzzahlige Programme zu entwickeln. Wir betrachten die zugrunde liegenden Algorithmen nicht.

Binärvariablen oder ganzzahlige Variablen?

■ Sudokus lösen

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

Naiver Ansatz

Wir verwenden 81 Integer Variablen

$$1 \leq y_i \leq 9.$$

Und jetzt?

■ Faustregel

- Verwenden Sie allgemeine Ganzzahlen (Integers), wenn sie tatsächliche Mengen darstellen und die Reihenfolge wichtig ist!
- Verwenden Sie Binärzahlen ($\{0, 1\}$ für jeden möglichen Wert einer Ganzzahl), wenn die Ganzzahlen konzeptuell nur „einige verschiedene Werte“ darstellen!

Beispiel: SEND + MORE = MONEY mittels Integer Programmierung

Problembeschreibung: SEND+MORE=MONEY[2]

- Klassisches Problem der Kryptographie
- Jeder Buchstabe repräsentiert eine Ziffer von 0 bis 9
- Keine Ziffer darf doppelt vorkommen

```
  S E N D
+ M O R E
-----
M O N E Y
```

- Welcher Buchstabe steht für welchen Wert?

[2] Mit Hilfe von (Mixed-)Integer-Programmierung lässt sich dieses Problem schnell lösen.

$$\begin{array}{ccccccc}
S_0 & + & S_1 & + & \dots & + & S_9 & = & 1 \\
+ & & + & & \dots & & + & & \\
E_0 & + & E_1 & + & \dots & + & E_9 & = & 1 \\
\vdots & & \vdots & & \vdots & & \vdots & & \\
+ & & + & & \dots & & + & &
\end{array}$$

3

„Optimierungsziel“

$$\begin{aligned}
& \sum_{i=0}^9 i \cdot S_i \times 1000 + \sum_{i=0}^9 i \cdot E_i \times 100 + \sum_{i=0}^9 i \cdot N_i \times 10 + \sum_{i=0}^9 i \cdot D_i \times 1 + \\
& \sum_{i=0}^9 i \cdot M_i \times 1000 + \sum_{i=0}^9 i \cdot O_i \times 100 + \sum_{i=0}^9 i \cdot R_i \times 10 + \sum_{i=0}^9 i \cdot E_i \times 1 = \\
& \sum_{i=0}^9 i \cdot M_i \times 10000 + \sum_{i=0}^9 i \cdot O_i \times 1000 + \sum_{i=0}^9 i \cdot N_i \times 100 + \sum_{i=0}^9 i \cdot E_i \times 10 + \sum_{i=0}^9 i \cdot Y_i \times 1
\end{aligned}$$

4

Umsetzung in Python mit Hilfe von *PuLP* <<https://coin-or.github.io/pulp/>>

Imports

```

1 from pulp import (
2     LpProblem, LpVariable, LpBinary,
3     lpSum
4 )

```

Variablen

```

1 VALS = range(10)
2 LETTERS = ["S", "E", "N", "D", "M", "O", "R", "Y"]
3
4 prob = LpProblem("SendMoreMoney") # Hier ist kein Optimierungsziel anzugeben.
5
6 choices = LpVariable.dicts("Choice", (LETTERS, VALS), cat=LpBinary)

```

5

Nebenbedingungen

```

1 # Jeder Buchstabe muss einen Wert haben
2 for l in LETTERS:
3     varsOfLetter = [choices[l][i] for i in VALS]
4     prob += lpSum(varsOfLetter) == 1
5
6 # Jeder Wert (0..9) darf nur einmal vorkommen
7 for i in VALS:
8     varsOfValue = [choices[l][i] for l in LETTERS]
9     prob += lpSum(varsOfValue) <= 1

```

„hauptsächliche Nebenbedingung“

```

1 prob += (
2     lpSum([i*choices["S"][i] for i in range(10)]) * 1000
3     + lpSum([i*choices["E"][i] for i in range(10)]) * 100
4     + lpSum([i*choices["N"][i] for i in range(10)]) * 10
5     + lpSum([i*choices["D"][i] for i in range(10)])
6     + lpSum([i*choices["M"][i] for i in range(10)]) * 1000
7     + lpSum([i*choices["O"][i] for i in range(10)]) * 100
8     + lpSum([i*choices["R"][i] for i in range(10)]) * 10
9     + lpSum([i*choices["E"][i] for i in range(10)])
10    == lpSum([i*choices["M"][i] for i in range(10)]) * 10000
11    + lpSum([i*choices["O"][i] for i in range(10)]) * 1000
12    + lpSum([i*choices["N"][i] for i in range(10)]) * 100
13    + lpSum([i*choices["E"][i] for i in range(10)]) * 10
14    + lpSum([i*choices["Y"][i] for i in range(10)]))

```

7

Lösung berechnen lassen

```

1 prob.solve()
2 values = [
3     c + "=" + str(i)
4     for c in choices
5     for i in range(10)
6     if choices[c][i].value() == 1
7 ]
8 print("; ".join(values))

```

Ausgabe

S= 8; E= 3; N= 2; D= 4; M= 0; O= 9; R= 1; Y= 7

Code: https://delors.github.io/theo-algo-mixed_integer_programming/code/send_more_money.py

8

Eine Formulierung wie $28 \leq S + E + N + D + M + O + R + Y \leq 44$, um sicherzustellen, dass (zumindest einige) Variablen nicht 0 sind stellt leider nicht die gewünschte Nebenbedingung sicher, dass jeder Wert nur einmal vergeben wird ($0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 = 28$ und $0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45$).

Eine Lösung mit obiger Nebenbedingung wäre zum Beispiel:

S=0 E=8 N=9 D=0

M=0 O=0 R=9 E=8

M=0 O=0 N=9 E=8 Y=8

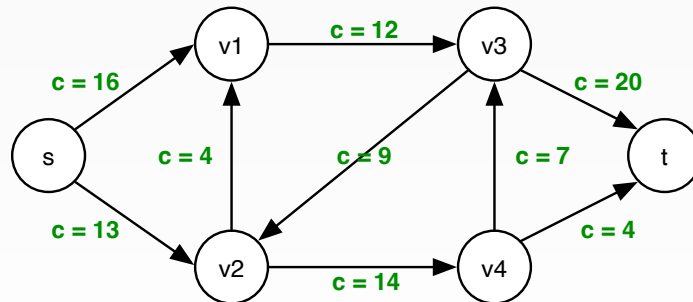
PuLP (Details)

Die Datenstruktur `choices` ist ein Dictionary mit folgendem Aufbau:

```
choices =  
    {'S': {0: Choice_S_0, 1: Choice_S_1,..., 9: Choice_S_9},  
     ...  
     'Y': {0: Choice_Y_0, 1: Choice_Y_1,..., 9: Choice_Y_9}}  
# choices['S'][0].name = 'Choice_S_0'
```

Maximaler Fluss

Berechnen Sie für folgenden Graphen den maximalen Fluss mit Hilfe von Pulp. Der Graph ist in der Vorlage definiert und kann als Grundlage für das Lösen des Problems verwendet werden. Orientieren Sie sich an dem Programm, dass sie im Vorfeld für das *Maximum-Flow-Problem* erstellt haben.



Vorlage

```

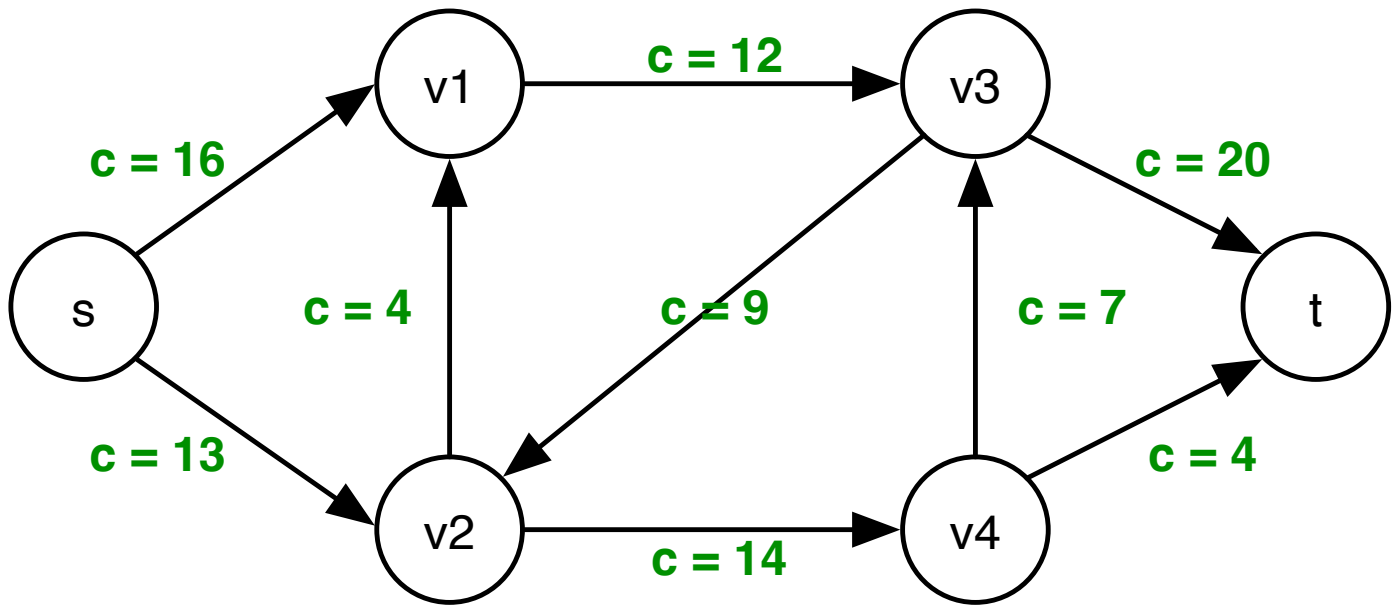
from pulp import (
    LpProblem,
    LpVariable,      # Erzeugt eine Variable
    LpMaximize,
    LpStatus,
    value,          # Gibt den Wert einer Variablen zurück
    lpSum,
)

CAPACITIES = {      # Kapazitäten der Kanten des Netzwerks
    "S": {"V1": 16, "V2": 13},
    "V1": {"V3": 12},
    "V2": {"V1": 4, "V4": 14},
    "V3": {"V2": 9, "T": 20},
    "V4": {"V3": 7, "T": 4},
    "T": {},
}

prob = LpProblem("Maximum Flow", LpMaximize)
  
```

Maximaler Fluss

Berechnen Sie für folgenden Graphen den maximalen Fluss mit Hilfe von Pulp. Der Graph ist in der Vorlage definiert und kann als Grundlage für das Lösen des Problems verwendet werden. Orientieren Sie sich an dem Programm, dass sie im Vorfeld für das *Maximum-Flow-Problem* erstellt haben.



■ Gruppenzuteilung

Finden Sie eine sehr gute Aufteilung von Personen (Studierenden) auf eine feste Anzahl an Gruppen, basierend auf den Präferenzen der Personen. Nutzen Sie dazu Mixed-Integer-Programmierung. Im Template ist eine initiale Aufgabenstellung hinterlegt, die es zu lösen gilt: Verteilung von 16 Studierenden auf 4 Gruppen inkl. Bewertungsmatrix (jeder Studierende hat jeden anderen mit Werten von 1 bis 10 bewertet). Ggf. ist die Funktion *pulp.allcombinations* beim Modellieren hilfreich.

https://delors.github.io/theo-algo-mixed_integer_programming/code/group_assignment_template.py

■ Alle Gruppen gleich glücklich machen

Fragen Sie sich was Sie tun müssten, wenn Sie zusätzlich sicherstellen wollen, dass alle Gruppen in etwa die gleiche Glücklichkeit haben sollen. (Hier geht es nur um ein Gedankenexperiment.)

Gruppenzuteilung

Finden Sie eine sehr gute Aufteilung von Personen (Studierenden) auf eine feste Anzahl an Gruppen, basierend auf den Präferenzen der Personen. Nutzen Sie dazu Mixed-Integer-Programmierung. Im Template ist eine initiale Aufgabenstellung hinterlegt, die es zu lösen gilt: Verteilung von 16 Studierenden auf 4 Gruppen inkl. Bewertungsmatrix (jeder Studierende hat jeden anderen mit Werten von 1 bis 10 bewertet). Ggf. ist die Funktion *pulp.allcombinations* beim Modellieren hilfreich.

https://delors.github.io/theo-algo-mixed_integer_programming/code/group_assignment_template.py

Alle Gruppen gleich glücklich machen

Fragen Sie sich was Sie tun müssten, wenn Sie zusätzlich sicherstellen wollen, dass alle Gruppen in etwa die gleiche Glücklichkeit haben sollen. (Hier geht es nur um ein Gedankenexperiment.)

Klausurvorbereitung

- Finden Sie eine Formulierung für das Lösen von Sudokus mittels Mixed-Integer-Programmierung.

Nächste Schritte

- Studiere Mathematische Programmiersprachen (AMPL, [ZIMPL](#) .)
- Studiere verfügbare Bibliotheken zum Lösen entsprechender Probleme (GLPK, SCIP, [Gurobi](#), [CPLEX](#), ...)

Hinweis

PuLP ist ein einfaches, aber mächtiges Werkzeug, um lineare und gemischt-ganzzahlige Programme zu beschreiben. PuLP nutzt im Hintergrund verschiedene Solver!