Abteilung Algorithmik **Winter 2020/21**
Institut für Betriebssysteme und Rechnerverbund
TU Braunschweig

Dr. Linda Kleist
Phillip Keldenich
Dominik Krupke

Mathematische Methoden der Algorithmik
Übungsblatt 4 vom 2. Dezember 2020

Die Abgabe eurer Lösungen zu diesem Blatt muss bis zum 16. Dezember 2020 um 16:00 Uhr per E-Mail an `sven.langner@tu-bs.de` erfolgen. Am besten gebt ihr eure Lösung als einzelnes PDF-Dokument (LaTeX, Word oder Scan) ab. Ihr könnt auch handschriftliche Lösungen scannen oder abfotografieren; nach Möglichkeit fasst ihr auch in diesem Fall am besten die einzelnen Seiten zu einem PDF-Dokument zusammen. Achtet dabei bitte einerseits auf Lesbarkeit und andererseits auf eine vernünftige Dateigröße (höchstens etwa 2 MB/Seite). Vermeidet bitte auch das Versenden von ZIP-Archiven, da es hier möglicherweise zu Problemen mit dem Virenscanner auf dem Mailserver der TU kommen kann.

Aufgabe 1 (Prüfungsfragen): Die kleine Übung soll auch als kontinuierliche Vorbereitung auf die Prüfung dienen, indem jeweils mögliche Prüfungsfragen zu den aktuellen Themen diskutiert werden.

Überlegt euch in eurer Hausaufgabengruppe 2 leichte, 2 mittlere und 2 schwere mögliche Prüfungsfragen zum Stoff der Vorlesungen 6–8 und gebt diese mit eurer Abgabe ab. Die Fragen werden bis zur kleinen Übung zusammengetragen und die besten Fragen werden dann kollaborativ in der kleinen Übung diskutiert. **(2+2+2 Punkte)**

Aufgabe 2 (Naiver LP-Solver mit Fundamentalsatz): Implementiere mit Python und NumPy einen Algorithmus, der lineare Programme der Form

$$(\mathrm{P}) : \min_{x \in \mathbb{R}^n} c^T x \text{ s.t. } Ax = b, x \geq 0$$

für A mit vollem Rang optimal lösen kann, indem er sämtliche möglichen Basen ausprobiert und die entsprechenden Basislösungen bestimmt. Orientiere dich bei der Implementierung an dem Codegerüst `naive_lp.py`, das im Material-Ordner in der nextcloud des IBR zur Verfügung steht bzw. zusammen mit dem Übungsblatt über die Mailingliste verschickt wurde oder hinten in diesem PDF gefunden werden kann. In diesem Codegerüst müssen nur drei Methoden angepasst werden.

Warum wird immer eine optimale Lösung gefunden, wenn eine existiert? Was passiert, wenn das gegebene LP unbeschränkt ist? **(25 Punkte)**

Aufgabe 3 (Verhältnisse als Zielfunktion): Seien $A \in \mathbb{R}^{m \times n}$ eine Matrix und $b \in \mathbb{R}^m, c \in \mathbb{R}^n, d \in \mathbb{R}^n$ Vektoren. Betrachte das Optimierungsproblem

$$(\mathrm{Q}) : \max_{x \in \mathbb{R}^n} \frac{\sum_{i=1}^n c_i x_i}{\sum_{i=1}^n d_i x_i} \text{ s.t. } Ax \leq b, \ \sum_{i=1}^n d_i x_i > 0, \ x \geq 0.$$

Finde zu (Q) ein LP (P) mit $\mathcal{O}(m)$ Nebenbedingungen und $\mathcal{O}(n)$ Variablen, aus dessen Lösung sich in $\mathcal{O}(n)$ Schritten die Lösung von (Q) ablesen lässt. Falls (Q) keine optimale Lösung hat, soll sich dies auch aus der Lösung von (P) ableiten lassen.

Hinweis: Ersetze zunächst $\frac{1}{\sum_{i=1}^n d_i x_i}$ durch eine Variable t und multipliziere die Zielfunktion und alle Nebenbedingungen mit t. **(15 Punkte)**

Aufgabe 4 (Simplex-Algorithmus): Nutze den Simplex-Algorithmus, um die folgenden LPs zu lösen. Verwende dafür die in der Vorlesung vorgestellte Tableaudarstellung. Argumentiere in jedem Schritt kurz, warum du die gewählte Nichtbasisvariable in die Basis aufnimmst bzw. die gewählte Basisvariable aus der Basis entfernst.

$$\begin{array}{lrcrcrcr} \min & & -x_1 & + & x_2 & - & 2x_3 & \\ \text{s.t.} & - & 2x_1 & - & x_2 & + & 3x_3 & \geq & -40 \\ & & x_1 & & & + & x_3 & \leq & 25 \\ & & & & x_2 & + & 3x_3 & \leq & 30 \\ & & x_1 & , & x_2 & , & x_3 & \geq & 0 \end{array}$$

$$\begin{array}{lrcrcrcrl} \max & 2x_1 & + & 4x_2 & + & 3x_3 & + & x_4 & \\ \text{s.t.} & 3x_1 & + & x_2 & + & x_3 & + & 4x_4 & \leq 12 \\ & x_1 & - & 3x_2 & + & 2x_3 & + & 3x_4 & \leq 7 \\ & 2x_1 & + & x_2 & + & 3x_3 & - & x_4 & \leq 10 \\ & & & & & x_1, & \ldots, & x_4 & \geq 0 \end{array}$$

(14 Punkte)

Code-Grundgerüst für Aufgabe 2

```python
# NumPy stellt u.a. Matrixen, Vektoren und
# Methoden zum Lösen von LGS (np.linalg.solve)
# und zur Bestimmung von Inversen (np.linalg.inv)
# zur Verfügung.
import numpy as np
import math

# combinations erlaubt über alle k-elementigen Teilmengen
# einer gegebenen Menge zu iterieren
from itertools import combinations
from typing import List


# Diese Methode muss nicht angepasst werden.
def select_columns(A: np.ndarray, columns: List[int]):
    """
    Bestimme die Submatrix, die aus A entsteht, wenn man nur die
```

```
    gegebenen Spaltenindizes nimmt.
    :param A: Eine Matrix.
    :param columns: Eine Liste von Spaltenindizes.
    :return:
    """
    return A[:, columns]



# Diese Methode muss nicht angepasst werden.
def add_nonbasic_variables(A: np.ndarray, x: np.ndarray,
                           columns: List[int]):
    """
    Füge zu einer Lösung für die Basisvariablen
    0-Einträge für die Nichtbasisvariablen hinzu.
    :param A: Die Matrix.
    :param x: Die Lösung für die Basisvariablen.
    :param columns: Die Spaltenindizes der Basisvariablen.
    :return:
    """
    result = np.zeros(A.shape[1], dtype=float)
    result[list(columns)] = x
    return result


# Diese Methode muss nicht angepasst werden.
def solve_linear_equations(A, b):
    """
    Löse das lineare Gleichungssystem Ax = b.
    :param A: Eine Matrix A.
    :param b: Ein Vektor b.
    :return: Den Lösungsvektor x, falls er existiert, sonst None.
    """
    try:
        return np.linalg.solve(A,b)
    except np.linalg.LinAlgError:
        return None

# Diese Methode soll angepasst werden.
# Muss hier eine Inverse bestimmt werden, oder geht es auch schneller?
def compute_basic_solution(A: np.ndarray, columns: List[int],
                           b: np.ndarray):
    """
    Diese Methode soll eine Basislösung zu den Spalten
    der gegebenen Indizes zur gegebenen Matrix A
    bestimmen, falls eine existiert. Andernfalls
    soll die Methode None zurückgeben.
    :param A: Eine zwei-dimensionale Matrix.
    :param columns: Die Spaltenindizes der Basis,
                    zu der die Basislösung bestimmt werden soll.
    :param b: Der b-Vektor.
    :return: None, falls keine Basislösung existiert.
             Andernfalls soll eine Basislösung zurückgegeben werden,
             die auch die Nicht-Basisvariablen enthält.
    """
    pass
```

```python
# Diese Methode soll angepasst werden.
def basic_solution_is_feasible(x: np.ndarray):
    """
    Diese Methode soll für eine gegebene Basislösung überprüfen,
    ob sie zulässig ist.
    :param x:
    :return:
    """
    pass


# Diese Methode soll angepasst werden.
def feasible_basic_solutions(A, b):
    """
    Gebe eine Sequenz aller zulässigen Basislösungen zurück,
    indem alle potentiellen Basen ausprobiert werden.
    """
    num_rows = A.shape[0]
    all_column_indices = range(A.shape[1])
    for basic_indices in ???:
        x = compute_basic_solution(A, basic_indices, b)
        if x is not None and basic_solution_is_feasible(x):
            yield x


# Diese Methode muss nicht angepasst werden.
def naive_solve_lp(A: np.ndarray, b: np.ndarray, c: np.ndarray):
    """
    Löst das gegebene LP min c^Tx s.t. Ax = b, x >= 0,
    indem alle potentiellen Basen ausprobiert werden.
    :param A: Die Matrix A.
    :param b: Der Vektor b.
    :param c: Der Vektor c.
    :return: (x*, v), wobei x* die beste Lösung und v ihr Lösungswert ist.
    """
    # @ ist der Operator für Matrix-Multiplikation, c.T transponiert c.
    value_of = lambda x: c.T @ x
    best_x = min(feasible_basic_solutions(A, b),
                 default=None, key=value_of)
    return None if best_x is None else\
            (best_x, value_of(best_x))
```