Министерство образования Российской Федерации

МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
им. Н.Э. Баумана

Факультет: Информатика и системы управления
Кафедра: Информационная безопасность (ИУ8)

ТЕОРИЯ ПРИНЯТИЯ РЕШЕНИЙ В УСЛОВИЯХ ИНФОРМАЦИОННЫХ КОНФЛИКТОВ

Лабораторная работа №1 на тему:
«Симплекс-метод решения задач линейного программирования»

Вариант 3

Преподаватель:
Коннова Н.С.

Студент:
Андреев Г.С.

Группа:
ИУ8-71

Москва 2021

Цель работы
Изучение симплекс-метода решения задачи линейного программирования (ЛП).

Постановка задачи
Требуется найти решение следующей задачи линейного программирования (ЛП):

$$F = cx \rightarrow max,$$

$$Ax \le b,$$

$$x \ge 0.$$

Здесь $x = [x_1, x_2, x_3]^T$ – искомый вектор решения;

`c = [3 3 7]` – вектор коэффициентов *целевой функции* (ЦФ) F;

```
A = [
  [1  1  1]
  [1  4  0]
  [0 1/2 3]
]
```
– матрица системы ограничений;

`b = [3 5 7]` – вектор правой части системы ограничений.

В развернутой форме задача имеет вид $F = \sum_{i=1}^{n} c_i x_i \rightarrow max,$

$x_i \ge 0, i = 1, \ldots, n.$

Каноническая форма записи задачи ЛП
Используя фиктивные переменные x_4, x_5, x_6 приведем исходную задачу к каноническому виду

$$F = -(3x_1 + 3x_2 + 7x_3) \rightarrow min,$$

$$\begin{cases} x_4 = 3 - (x_1 + x_2 + x_3) \\ x_5 = 5 - (x_1 + 4x_2) \\ x_6 = 7 - (0.5x_2 + 3x_3) \end{cases}$$

$$x_i \ge 0$$

Исходная симплекс таблица

	S0	X1	X2	X3
X4	3	1	1	1
X5	5	1	4	0
X6	7	0	0.5	3
F	0	-3	-3	-7

Промежуточные симплекс таблицы

Так как в столбце свободных членов отсутствуют отрицательные элементы то исходная симплекс таблица является опорным решением, поэтому приступим к поиску оптимального решения.

1) Индекс разрешающего элемента (x4, x1):

	S0	X4	X2	X3
X1	3	1	1	1
X5	2	-1	3	-1
X6	7	0	0.5	3
F	9	3	0	-4

Значение исходных переменных для данного решения:

$$\begin{cases} x_1 = 3 \\ x_2 = 0 \\ x_3 = 0 \end{cases}$$

Значение ЦФ для данных переменных:

$F = 9$

2) Индекс разрешающего элемента (x6, x3):

	S0	X4	X2	X6
X1	0.6667	1	0.8333	-0.3333
X5	4.3333	-1	3.1667	0.3333
X3	2.3333	0	0.1667	0.3333
F	18.3333	3	0.6667	1.3333

Значение исходных переменных для данного решения:

$$\begin{cases} x_1 = 0.667 \\ x_2 = 0 \\ x_3 = 2.333 \end{cases}$$

Значение ЦФ для данных переменных:

$F = 18.3333$

Так как в последней строке нет отрицательных коэффициентов, то данное решение является оптимальным

Проверка

Подставив полученные значения, можно убедиться в правильности решения:

$F(x_1, x_2, x_3) = 0.667 * 3.0 + 0 * 3.0 + 2.333 * 7.0 = 18.333$ – проверка значения целевой функции

$$\text{Условие } 1: 0.667 * 1 + 0 * 1 + 2.333 * 1 = 3.0 \le 3.0$$
$$\text{Условие } 2: 0.667 * 1 + 0 * 4 + 2.333 * 0 = 0.667 \le 5.0$$
$$\text{Условие } 3: 0.667 * 0 + 0 * 0 + 2.333 * 3 = 7.0 \le 7.0$$

Вывод

В ходе работы был изучен симплекс-метод решения задач линейного программирования. Основная процедура которого заключается в заменах переменных базиса, что сводится к перерасчету коэффициентов в симплекс-таблицах, таким образом данная процедура легко формализуется для вычисления данного метода с помощью ЭВМ.

Листинг программы

Файл simplex_table.py (https://github.com/andreev-g/iu8-decision-theory/blob/master/src/simplex/simplex_table.py)

```
import typing as t
import pandas as pd
from tabulate import tabulate

from src.simplex.simplex_problem import (
    FuncTarget,
    SimplexProblem,
    HUMAN_COMP_SIGNS
)


class SimplexTable(pd.DataFrame):

    _F = "F"
    _Si0 = "si0"
    _ROW = "row"
    _COL = "column"

    NO_SOLUTIONS_ERR_MSG = "there aren't solutions"

    _problem: SimplexProblem = None

    def __init__(
            self,
            problem: SimplexProblem
    ):
        self._problem = problem.copy()
        canonical_matrix = problem.get_canonical()
        minor_vars_num = len(canonical_matrix[0]) - 1
        basis_vars_num = len(canonical_matrix) - 1
        columns = [self._Si0] + [
            f"x{i}"
            for i in range(1, minor_vars_num + 1)
        ]
        index = [
            f"x{i + minor_vars_num}"
            for i in range(1, basis_vars_num + 1)
        ] + [self._F]
        super().__init__(
            data=canonical_matrix,
            index=index,
            columns=columns,
            dtype=float,
            copy=True
        )

    def find_base_solution(
```

```
            self,
            inplace: bool = False,
            print_logs: bool = False
    ) -> 'SimplexTable':
        simplex: SimplexTable = self._get_self(make_copy=not inplace)
        while True:
            if simplex.is_base_solution():
                break
            row, col = simplex._get_pivot_indices()
            simplex._swap_vars(row, col)
            if print_logs:
                print()
                print("~" * 70 + "\n")
                print(f"Разрешающие (строка, столбец) : ({row} , {col})")
                simplex.print()
        return simplex

    def find_optimal_solution(
            self,
            inplace: bool = False,
            print_logs: bool = False
    ) -> 'SimplexTable':
        simplex = self._get_self(make_copy=not inplace)
        while True:
            if simplex._is_optimal_solution():
                break
            row, col = simplex._get_pivot_indices(start_row=self._F)
            simplex._swap_vars(row, col)
            if print_logs:
                print(f"Разрешающие (строка, столбец) : ({row} , {col})")
                simplex.print()
                print()
                print("~" * 70 + "\n")
        return simplex

    def print(self) -> None:
        print(
            tabulate(
                self.applymap(lambda x: x if x != 0 else 0.),
                headers="keys",
                tablefmt="psql"
            )
        )

    def _swap_vars(self, row: str, col: str) -> None:
        self._check_swap_index(row, loc=self._ROW)
        self._check_swap_index(col, loc=self._COL)

        s_rk = 1 / self.loc[row, col]
        s_rj = self.loc[row] / self.loc[row, col]
        s_ik = -1 * self.loc[:, col] / self.loc[row, col]
        self.loc[:, :] = [
            [
                self.iloc[i, j] - self.loc[:, col].iloc[i] * self.loc[row].iloc[j] / self.loc[row, col]
                for j in range(len(self.columns))
            ]
            for i in range(len(self.index))
        ]
        self.loc[row, :] = s_rj
        self.loc[:, col] = s_ik
        self.loc[row, col] = s_rk
```

```
        self.rename(columns={col: row}, inplace=True)
        self.rename(index={row: col}, inplace=True)

    def _check_swap_index(self, name: str, loc: str) -> None:
        if loc not in (self._ROW, self._COL):
            raise ValueError(f"please, specify one of (\"{self._ROW}\", \"{self._COL}\"); passed:
{loc}")
        sequence = self.columns if loc == self._COL else self.index
        if name not in sequence:
            raise IndexError(f"No such value in {loc}: {name}")
        if name in (self._F, self._Si0):
            raise ValueError(f"Not allowed to access {loc} value: {name}")

    def is_base_solution(self) -> bool:
        for row in self.index.copy().drop(self._F):
            if self.loc[row, self._Si0] < 0:
                assert any(self.loc[row].iloc[1:] < 0), self.NO_SOLUTIONS_ERR_MSG
                return False
        return True

    def _is_optimal_solution(self) -> bool:
        if self._problem.target == FuncTarget.MIN:
            return all(self.loc[self._F] < 0)
        return all(self.loc[self._F] > 0)

    def _get_pivot_indices(self, start_row: str = None) -> t.Tuple[str, str]:
        if not start_row:
            for st_row in self.index:
                if self.loc[st_row, self._Si0] < 0:
                    start_row = st_row
                    break

        if self._problem.target == FuncTarget.MIN:
            col = self.loc[start_row].drop(self._Si0).idxmax()
            if self.loc[start_row, col] < 0:
                raise ValueError

        else:
            col = self.loc[start_row].drop(self._Si0).idxmin()
            if self.loc[start_row, col] > 0:
                raise ValueError

        row = None
        row_value = None
        for row_name in self.index.drop(self._F):
            if self.loc[row_name, self._Si0] != 0 and self.loc[row_name, col] != 0:
                calc_value = self.loc[row_name, self._Si0] / self.loc[row_name, col]
                if row is None or calc_value < row_value:
                    row = row_name
                    row_value = calc_value
        if row is None:
            raise ValueError

        return row, col

    def check_solution(self) -> bool:
        solution = self.get_solution()
        simplex_f = round(self.loc[self._F, self._Si0], 3)
        calculated_f = round(sum(solution[i] * self._problem.c[i] for i in
```

```
range(len(self._problem.c))), 3)
        print("F: " + " + ".join(
            f"{round(solution[i], 3)} * {round(self._problem.c[i], 3)}" for i in
range(len(self._problem.c))
        ) + f" == {simplex_f}")
        for i, row in enumerate(self._problem.A):
            comp_sign = HUMAN_COMP_SIGNS[self._problem.comp_signs[i]]
            print(f"Условие {i + 1}: " + " + ".join(
                f"{round(solution[j], 3)} * {round(a)}" for j, a in enumerate(row)
            ) + f" == {round(sum(solution[j] * a for j, a in enumerate(row)), 3)} {comp_sign}
{self._problem.b[i]}")
        return simplex_f == calculated_f

    def get_solution(self) -> t.List[float]:
        return [
            0 if f"x{i}" not in self.index else self.loc[f"x{i}", self._Si0]
            for i in range(1, len(self))
        ]

    def _get_self(self, make_copy: bool) -> 'SimplexTable':
        if make_copy:
            return self.copy()
        return self
```