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 КОНФЛИКТАХ

Лабораторная работа №3 на тему:
«Формулировка и решение ЦЗЛП методом ветвей и границ»

Вариант 3

Преподаватель: Коннова Н.С.
Студент: Андреев Г.С.
Группа: ИУ8-71

Москва 2021

Цель работы

Изучить постановку задачи целочисленного линейного программирования. Получить решение задачи ЦЛП методом отсекающих плоскостей.

Постановка задачи

Исходная задача ЦЛП выглядит следующим образом:

$$\begin{cases} F = cx \to max \\ Ax \le b \\ x_i \ge 0, \\ x_i - целое, i = \overline{1,3} \end{cases}$$

Где:

$x = (x_1 \quad x_2 \quad x_3)^T$ – искомый вектор решения;

$c = (2 \quad 6 \quad 7)$ – вектор коэффициентов *целевой функции* (ЦФ) F;

$A = \begin{pmatrix} 3 & 1 & 1 \\ 1 & 2 & 0 \\ 0 & 0.5 & 1 \end{pmatrix}$ – матрица системы ограничений;

$b = (3 \quad 8 \quad 1)^T$ – вектор правой части системы ограничений.

Предлагается найти решение данной задачи ЦЛП отсекающих плоскостей.

Решение полным перебором

Метод полного перебора гарантирует нахождение оптимального решения. Однако он является крайне трудоемким в вычислительном плане, что делает его применение нежелательным.

Методом полного перебора было обнаружено 5 решений задачи ЦЛП:

$x^* = (0 \quad 0 \quad 0), F(x^*) = 0$

$x^* = (0 \quad 0 \quad 1), F(x^*) = 7$

$x^* = (0 \quad 1 \quad 0), F(x^*) = 6$

$x^* = (0 \quad 2 \quad 0), F(x^*) = 12$

$x^* = (1 \quad 0 \quad 0), F(x^*) = 2$

Из них оптимальным решением является решение: $x^* = (0 \quad 2 \quad 0), F(x^*) = 12$

Решение ЗЦЛП методом отсекающих плоскостей

Решим исходную задачу ЦЛП как задачу ЛП с помощью симплекс-метода:

Исходная симплекс-таблица:

	Si0	x1	x2	x3	x4	x5	x6
x4	3	3	1	1	1	0	0
x5	8	1	2	0	0	1	0
x6	1	0	0.5	2	0	0	1
F	0	-2	-6	-7	0	0	0

Итоговая симплекс-таблица:

	Si0	x1	x2	x3	x4	x5	x6
x1	0.333	1	0	-1	0.333	0	-0.667
x5	3.667	0	0	-7	-0.333	1	-3.333
x2	2	0	1	4	0	0	2
F	12.667	0	0	15	0.667	0	10.667

Найдено решение: $x^* = (0.3333 \quad 2 \quad 0), F(x^*) = 12.6667$

Найденое решение не является целочисленным, потому добавим новое ограничение (отсекающую плоскость) в исходную задачу.

Для получения нового ограничения, необходимо найти переменную с наибольшей нецелой частью. В данном случае, этой переменной будет являться x_1, поскольку x_1 - единственная нецелая переменная в решении. Составим ограничение:

$$\{1\}x_1+\{0\}x_2+\{-1\}x_3+\{\frac{1}{3}\}x_4+\{0\}x_5+\{-\frac{2}{3}\}x_6=\{\frac{1}{3}\}$$

$$\frac{1}{3}x_4+\frac{1}{3}x_6\geq\frac{1}{3}\Leftrightarrow x_4+x_6\geq 1$$

И выразим это ограничение через существенные переменные:

$$\begin{cases} x_4+x_6\geq 1 \\ x_4=3-(3x_1+x_2) \\ x_6=1-(\frac{1}{2}x_2+2x_3) \end{cases}\Leftrightarrow x_1+\frac{1}{2}x_2+\frac{2}{3}x_3\leq 1$$

Составим симплекс-таблицу, дополненную новым ограничением:

	Si0	x1	x2	x3	x4	x5	x6	x7
x4	3	3	1	1	1	0	0	0
x5	8	1	2	0	0	1	0	0
x6	1	0	0.5	2	0	0	1	0
x7	-0.333	0	0	0	-0.333	0	-0.333	1
F	0	-2	-6	-7	0	0	0	0

Решим дополненную задачу симплекс-методом и получим итоговую симплекс-таблицу:

	Si0	x1	x2	x3	x4	x5	x6	x7
x2	2	2	1	2	0	0	0	1
x5	4	-3	0	-4	0	0	1	-0.5
x6	0	-1	0	1	0	0	1	-0.5
x4	1	1	0	-1	1	0	0	-1
F	12	10	0	5	0	0	0	6

Найдено решение: $x^*=(0 \quad 2 \quad 0), F(x^*)=12$

Найдено решение: Найденое решение является целочисленным, потому запомним его. Перейдем к поиску оптимального решения из тех, что удалось найти, поскольку больше не осталось ветвей для дальнейших ветвлений. $x^*=(0 \quad 2 \quad 0), F(x^*)=12$ $x^*=(1 \quad 0 \quad 0), F(x^*)=2$. Заметим, что оно совпадает с решением найденым полным перебором. Данный факт дает основание пологать, что решение действительно является оптимальным.

Визуализация отсекающих плоскостей

Для того, чтобы ход решения задачи ЦЛП методом отсекающих плоскостей было легче понять, покажем как выглядят отсекающие плоскости, соответсвующие исходным и дополнительному ограничениям.

На нижеприведенном рисунке показаны исходные ограничения в виде плоскостей в трехмерном пространстве и решение задачи ЛП в виде точки B.

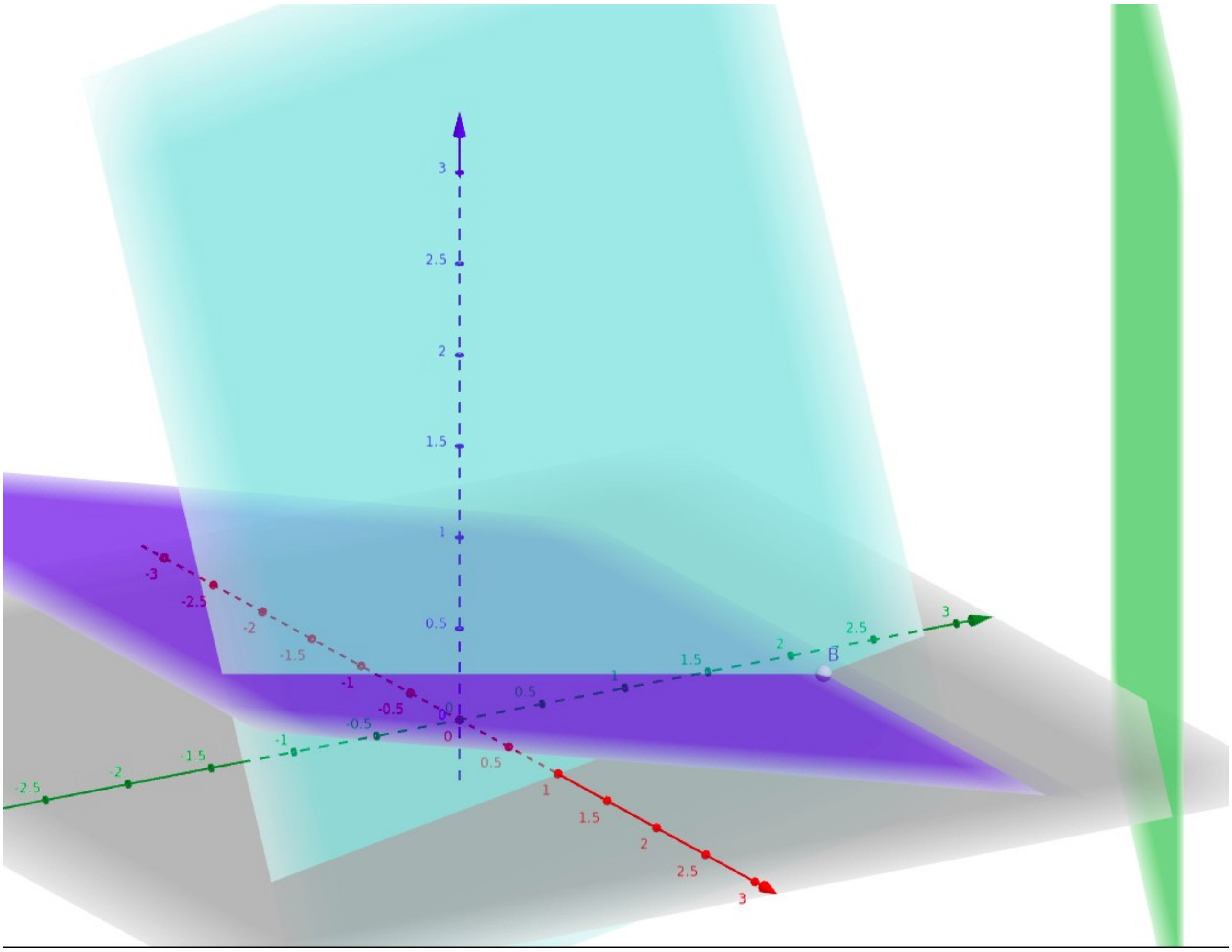

Рисунок 1. Визуализация решения задачи ЛП

Видно, что решение находится в экстримальной точке многогранника. Это соответствует действительности, ведь симплекс-метод ищет экстримальную из вершин многогранника ограничений путем обхода вершин.

Добавим найденное нами ограничение в виде плоскости красного цвета. И покажем все ограничения, точку B, соответствующую решению задачи ЛП, и точку A, соответствующую решению задачи ЦЛП.

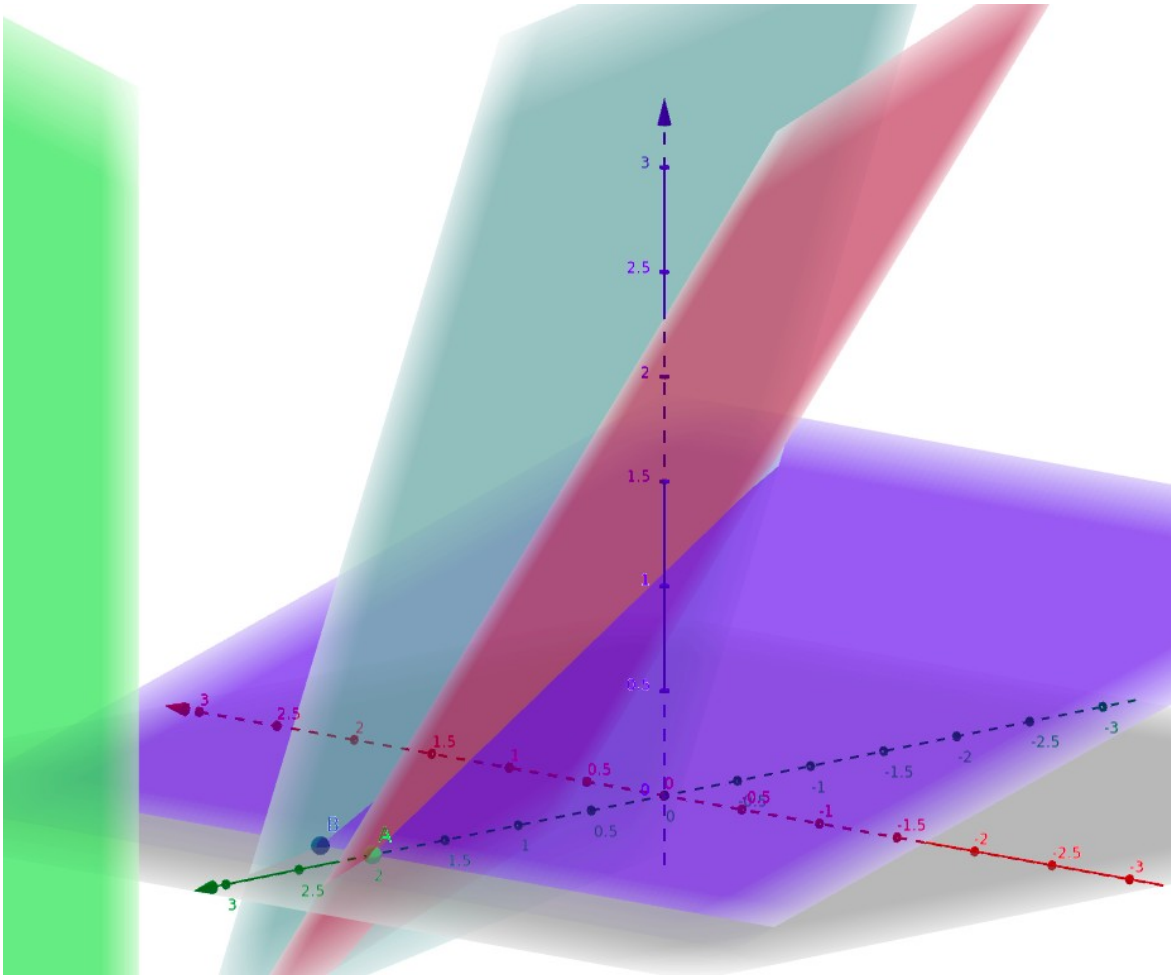

Рисунок 2. Визуализация решения задачи ЛП и ЦЛП

Целочисленное решение также попало в угол многогранника. Но в данном случае угол образован новой плоскостью, соответсвующей найденному нами ограничению. Обозначения:

	$A = (0, 2, 0)$
	eq1: $3x + y + z = 3$
	eq2: $x + 2y = 8$
	eq3: $0.5y + 2z = 1$
	eq4 : $x + \frac{1}{2}y + \frac{2}{3}z = 1$
	$B = \left(\frac{1}{3}, 2, 0\right)$ $\rightarrow (0.33, 2, 0)$

Рисунок 3. Обозначения

Метод округления переменных до ближайшего целого

Решим задачу ЦЛП симплекс-методом и округлим элементы вектора решений до ближайших целых.

$x^* = (0.3333 \quad 2 \quad 0), F(x^*) = 12.6667$

Округлим решение:

$x^* = (0 \quad 2 \quad 0), F(x^*) = 12$

Методом округления было найдено целочисленное решение, которое является оптимальным. Однако теортически метод округления не является корректным способом нахождения решения задач ЦЛП, хотя на данном конкретном примере решения совпали. Метод округления некорректен, поскольку система ограничений может вовсе не иметь решений при округлении переменных.

Вывод

В данной работе были изучены методы решения задачи ЦЛП. Конкретно были применены на практике методы полного перебора и метод отсекающих плоскостей (метод Гомори).

Листинг программы

Файл main.py:

```
import itertools
from typing import List, Tuple
import numpy
from numpy.lib import math
import pandas



def print_separator():
    print('-' * 50)



class SimplexMethod:
    def __init__(self, a, b, c, mode):
        self.A = numpy.array(a)
        self.b = numpy.array(b)
        self.c = numpy.array(c)
        self.mode = mode

        # распологаем вектор свободных коэффициентов слева от матрицы A
        matr = numpy.c_[b, self.A]
        # распологаем вектор коэффициентов целевой функции в нижней строке симплекс-
таблицы
        self.table = numpy.r_[matr, [[0, *self.c]]]
        # сохраняем исходную симплекс-таблицу для того, чтобы иметь возможность
        # проверить валидность решения в дальнейшем
        self.src_table = numpy.copy(self.table)
        # инвертируем знак коэффициентов целевой функции
        self.table[-1] *= -1

        # формируем легенды столбцев: [S0, x1, x2, x3]
        s0 = 'S0'
        self.columns = [s0] + [f'x_{i + 1}' for i in range(self.table.shape[1] - 1)]
        # формируем легенды строк: [F, x4, x5, x6]
        f = 'F'
        self.rows = [f'x_{i + 4}' for i in range(self.b.size)] + [f]

    def _exchange_basic_variable(self, resolving_element) -> numpy.ndarray:
        """
```

```
        Данный метод предназначен введения в базис переменной из столбца k вместо
        переменной из строки r, где r = resoling_element[0], k = resoling_element[1].
        Возвращаемым значением является симплекс-таблица с замененной базисной
переменной
        """
        # переименовываем легенду базисной переменной, которую хотим заменить
        self.rows[resolving_element[0]] = self.columns[resolving_element[1]]
        # создаем матрицу, в которую будем записывать новую симплекс-таблицу
        new_matrix = numpy.zeros(self.table.shape)

        r = resolving_element[0]
        k = resolving_element[1]
        # проходим по всем элементам старой таблицы и формируем новую
        for i, j in numpy.ndindex(self.table.shape):
            if i == r:
                # попали на разрешающую строку
                # формула: s*[r][j] = s[r][j]/s[r][k]
                new_matrix[r, j] = self.table[r, j] / self.table[r, k]
            else:
                # попали на любой другой элемент
                # формула: s*[i][j] = s*[i][j] - (s[i][k] * s[r][j])/s[r][k]
                prod = (self.table[i, k] * self.table[r, j] / self.table[r, k])
                new_matrix[i, j] = self.table[i, j] - prod

        return new_matrix

    def _to_pivot_solution(self) -> List[Tuple[Tuple[int, int], pandas.DataFrame]]:
        """
        Данный метод преобразует симплекс-таблицу до опороного решения (по ходу
        преобразования понимаем совместна ли система).
        Метод возвращает ход решения опорного решения в виде массива вида:
        [( (r_0, k_0), simplex-table_0), ..., ( (r_n, k_n)), simplex-table_n)]
        """


        def find_resolving_element() -> Tuple[int, int]:
            """
            Функция ищет разрешающий элемент.
            Функция возвращает кортеж индексов разрешающего элемента в случае
нахождения,
            иначе возвращает None
            """
```

```
        # получаем массив свободных коэффициентов (коэффициенты при ЦФ не
рассматриваем)
        free_coefs = self.table[:-1, 0]
        # ищем индекс строки, в которой свобоный коэффициент < 0
        negative_row = numpy.where(free_coefs < 0, free_coefs, numpy.inf).argmin()
        # если такой строки нет, то разрешающего элемента нет
        if negative_row == 0 and free_coefs[negative_row] > 0:
            return None
        # получаем коэффициенты при переменных из строки, где есть своб. коэф. < 0
        row = self.table[negative_row, 1:]

        # проверяем на наличие решений (если нет коэффициента < 0, то решений нет)
        assert numpy.any(row < 0), 'система несовместна'
        # находим разрешающий столбец (там должен находиться коэффициет < 0)
        k = numpy.where(row < 0, row, numpy.inf).argmin()
        # увеличиваем k на 1, чтобы учесть наличие свободного коэффициента в таблице
        k += 1
        # получаем коэффициенты при переменных для разрешающего столбца (без ЦФ)
        resolving_column = self.table[:-1, k]
        # игнорируем возможные деления на 0 в контекстном менеджере
        with numpy.errstate(divide='ignore'):
            # делим соответствующие свобдные коэффициенты на коэффициенты
            # разрешающего столбца и записываем бесконечность в ячейки, где частное
<= 0
            quotient = free_coefs / resolving_column
            quotient[quotient <= 0] = numpy.inf
            # берем индекс наименьшего положительного частного и проверяем, что он
            # действительно найден. В случае ненахождения считаем, что решений
бесконечно
            r = quotient.argmin()
            assert quotient[r] != numpy.inf, f'система имеет бесконечно число решений'

        return (r, k)

    # массив с шагами нахождения опорного решения
    solution_progress = list()
    # преобразуем симплекс-таблицу до тех пор, пока не будет найдено опорное
решение
    found_pivot_solution = False
    while not found_pivot_solution:
        # находим разрешающий элемент. Если его нет, то данное решение является
```

```
опорным
            rk = find_resolving_element()
            if rk is None:
                found_pivot_solution = True
                continue

            # производим замен базисной переменной и добавляем запись в протокол
решения
            self.table = self._exchange_basic_variable(rk)
            solution_progress.append((
                rk,
                pandas.DataFrame(
                    data=numpy.copy(self.table),
                    index=numpy.copy(self.rows),
                    columns=numpy.copy(self.columns)
                )
            ))

        return solution_progress

    def _find_optimal_solution(self) -> List[Tuple[Tuple[int, int], pandas.DataFrame]]:
        """
        Данный метод преобразует симплекс-таблицу до оптимального решения.
        Метод возвращает ход решения опорного решения в виде массива вида:
        [( (r_0, k_0), simplex-table_0), ..., ( (r_n, k_n)), simplex-table_n)]
        """

        def find_resolving_element() -> Tuple[int, int]:
            """
            Функция ищет разрешающий элемент.
            Функция возвращает кортеж индексов разрешающего элемента
            """
            k = 0
            if self.mode == 'min':
                # если мы минимизируем ЦФ, то ищем в коэффициентах ЦФ максимальный
                # в качестве разрешающего
                k = numpy.argmax(self.table[-1, :][1:]) + 1
            elif self.mode == 'max':
                # в случае максимизации - минимальный
                k = numpy.argmin(self.table[-1, :][1:]) + 1
            else:
```

```
            raise ValueError("mode could be 'max' or 'min' only")

        # получаем разрешающий столбец и стобец свободных коэффициентов
        resolving_column = self.table[:, k][:-1]
        free_coefs = self.table[:, 0][:-1]
        # игнорируем возможные деления на 0 в контекстном менеджере
        with numpy.errstate(divide='ignore'):
            # делим соответствующие свобдные коэффициенты на коэффициенты
            # разрешающего столбца и записываем бесконечность в ячейки, где частное
<= 0
            quotient = free_coefs / resolving_column
            quotient[quotient < 0] = numpy.inf
            # берем индекс наименьшего положительного частного и проверяем, что он
            # действительно найден. В случае ненахождения считаем, что решений
бесконечно
            r = quotient.argmin()
            assert quotient[r] != numpy.inf, f'Система имеет бесконечно много решений'
        return r, k

    # задаем условие остановки алгоритма: если минимизируем, то остановимся,
    # когда все коэффициенты при ЦФ <= 0, если максимизируем - все коэффициенты
при ЦФ >= 0
    if self.mode == 'min':
        stop_condition = lambda: not all(i <= 0 for i in self.table[-1, 1:])
    elif self.mode == 'max':
        stop_condition = lambda: not all(i >= 0 for i in self.table[-1, 1:])

    # массив с шагами нахождения оптимального решения
    solution_progress = list()
    while stop_condition():
        # находим разрешающий элемент
        rk = find_resolving_element()

        # производим замен базисной переменной и добавляем запись в протокол
решения
        self.table = self._exchange_basic_variable(rk)
        solution_progress.append((
            rk,
            pandas.DataFrame(
                data=numpy.copy(self.table),
                index=numpy.copy(self.rows),
```

```
                columns=numpy.copy(self.columns)
            )
        ))

    return solution_progress

def _verify_solution(self):
    """
    Данный метод проверяет правильно ли была вычислена ЦФ, а также смотрит не
    были ли были нарушены ограничения
    """
    # получаем решение задачи и коэффициенты при целевой функции
    solution = self._get_solution()
    f_coefs = self.src_table[-1, 1:4]
    # считаем значение целевой функции и сравниваем с найденным значением
    f = sum([f_coefs[idx] * var for idx, var in enumerate(solution[1:4])])
    assert solution[0] == f, f'Результат оптимального решения не совпадает с
коэффициентами F={solution[0]}, f={f}'

    # итерируемся по ограничениям и проверяем соответствует ли им решение
    for number, limitation_conditions in enumerate(self.src_table[:-1]):
        prod = limitation_conditions[1:] * solution[1:]
        limit = numpy.sum(prod)
        assert limitation_conditions[
                   0] >= limit, f'Ограничение №{number + 1} нарушено: {limit} <=
{limitation_conditions[0]}'
        print(f'Ограничение №{number + 1}: {limit} <= {limitation_conditions[0]}')

    print('Решение верно!')

def _get_solution(self) -> numpy.ndarray:
    """
    Данный метод возвращает решение задачи в виде:
    [ F(X), x1, ..., xn ]
    """
    # создаем массив коэффициентов и кладем на первое место значение ЦФ
    solution = [self.table[-1, 0]]
    # добавляем коэффициенты базисных переменных, остальные коэффициенты
полагаем равными нулю
    for var_number in range(1, self.table.shape[1]):
        if f'x_{var_number}' in self.rows:
```

```
            var = self.table[self.rows.index(f'x_{var_number}'), 0]
        else:
            var = 0
        solution.append(var)
    return numpy.array(solution)

def solve(self):
    def print_progress(progress):
        """
        Функция печати ряда шагов (хода решений) симплекс-метода
        """
        for step in progress:
            print_separator()
            print('Индекс разрешающего элемента: ', step[0])
            print(step[1])

    print('СИМПЛЕКС МЕТОД')
    print_separator()
    print('Исходная симплекс-таблица:')
    print(pandas.DataFrame(data=self.table, index=self.rows, columns=self.columns))

    progress = self._to_pivot_solution()
    print_progress(progress)

    print('Поиск оптимального решения:')
    progress = self._find_optimal_solution()
    print_progress(progress)

    self._verify_solution()
    solution = self._get_solution()

    print('x1 =', solution[1])
    print('x2 =', solution[2])
    print('x3 =', solution[3])
    print('F =', solution[0])

    return self.table[:, 0]



class Gomory:
```

```
    def __init__(self, a, b, c, mode):
        self.a = numpy.array(a)
        self.b = numpy.array(b)
        self.c = numpy.array(c)
        self.mode = mode

        # дополняем матрицу ограничений A и вектор ЦФ фиктивными переменными
        # A, E -> A|E
        self.a = numpy.column_stack((self.a, numpy.eye(self.b.size)))
        # for example: c, (0, 0, 0) -> (c0, ..., ci, 0, 0, 0)
        self.c = numpy.append(self.c, numpy.zeros(self.b.size))

        # создаем объект класса SimplexMethod в котором и будут производится
        # все вычисления симплекс-метода
        self.simplex = SimplexMethod(self.a, self.b, self.c, self.mode)

    @staticmethod
    def _is_integer_solution(solution):
        """
        Данная функция проверяет является ли это решение целочисленным
        """
        for el in solution:
            if not el.is_integer():
                return False
        return True

    @staticmethod
    def _find_with_max_fractional_part(solution):
        """
        Данная функция получает значение максимальной дробной части
        """
        return (solution % 1).argmax()

    def solve(self):
        print('МЕТОД ОТСЕКАЮЩИХ ПЛОСКОСТЕЙ (МЕТОД ГОМОРИ)')

        found_interger_solution = False
        while not found_interger_solution:
            # находим оптимальное решение симплекс-методом и проверяем на
целочисленность
```

```
            solution = self.simplex.solve()
            if self._is_integer_solution(solution[:-1]):
                found_interger_solution = True
                continue

            # ищем базисную переменную с наибольшей дробной частью
            idx = self._find_with_max_fractional_part(solution)
            # получаем массив ограничений, который состоит из дробныых частей всех
            # переменных в разложении переменной найденной на предыдущем шаге
            var_fractions = self.simplex.table[idx, 1:]
            var_fractions %= 1
            # дополняем снизу матрицу ограничений новым ограничением из дробных
            # частей, взятых сотрицательным знаком
            var_fractions *= -1
            a = numpy.vstack((self.simplex.A, var_fractions))
            # получаем значение свободного коэффициента при найденной переменной и
            # дополняем им вектор свобоных членов
            free_coef = -(solution[idx] % 1)
            b = numpy.append(self.simplex.b, free_coef)
            # создаем столбец для новой фиктивной переменной, соответсвующей
найденному ограничению
            # и дополняем им справа симплекс-таблицу
            dummy_col = numpy.zeros(b.size)
            dummy_col[-1] = 1
            a = numpy.column_stack((a, dummy_col))
            # дополняем вектор коэффициентов ЦФ до количества переменных
            c = numpy.append(self.simplex.c, 0)
            # запускам сиплекс-метод с округлением всех чисел до 10 знаков во избежание
зацикливания
            self.simplex = SimplexMethod(numpy.around(a, 10), numpy.around(b, 10),
numpy.around(c, 10), self.mode)

        return solution



def brute_force(a, b, c, optimum):
    """
    Функция полного перебора всех возможных целочисленных переменных
    """
    print_separator()
    print('МЕТОД ПОЛНОГО ПЕРЕБОРА')
```

```
    a = numpy.array(a)
    b = numpy.array(b)
    c = numpy.array(c)
    solutions = {}

    var_limit = optimum / numpy.min(c)
    for combination in itertools.product(numpy.arange(var_limit), repeat=c.size):
        number_of_valid_constraints = 0
        for i in range(b.size):
            constraints = a[i] * combination
            if numpy.sum(constraints) <= b[i]:
                number_of_valid_constraints += 1

        if number_of_valid_constraints == b.size:
            result = numpy.sum(combination * c)
            solutions[result] = combination
            print(combination, result)

    optimal_solution = max(solutions.keys())
    return optimal_solution, solutions[optimal_solution]



if __name__ == '__main__':
    A = [[-2, -6],
         [-8, -3]]
    c = [1, 1]
    b = [-1, -1]

    s = SimplexMethod(A, b, c, 'max')
    s.solve()

    A = [[3, 1, 1],
         [1, 2, 0],
         [0, 0.5, 2]]
    c = [2, 6, 7]
    b = [3, 8, 1]

    gomory_method = Gomory(A, b, c, 'max')
    solution = gomory_method.solve()
```

```
solution = brute_force(A, b, c, solution[-1])
print('Решение:')
print('x1 =', solution[1][0])
print('x2 =', solution[1][1])
print('x3 =', solution[1][2])
print('F =', solution[0])
```