Министерство науки и высшего образования Российской Федерации
Калужский филиал федерального государственного автономного образовательного учреждения высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(КФ МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ ИУК Информатика и управление

КАФЕДРА ИУК4 Программное обеспечение ЭВМ, информационные технологии

ЛАБОРАТОРНАЯ РАБОТА 3

«Задачи целочисленного линейного программирования»

по дисциплине: «*Методы принятия решений в программной инженерии*»

Выполнил: студент группы ИУК4-72Б _____ Губин Е.В.
(Подпись) (И.О. Фамилия)

Проверил: _____ Никитенко У.В.
(Подпись) (И.О. Фамилия)

Дата сдачи (защиты):

Результаты сдачи (защиты):

- Балльная оценка:
- Оценка:

Калуга, 2025

Цель работы: сформировать практические навыки анализа возможностей построения и выделения наиболее важных свойств объектов моделей для моделирования и использования специализированных программных пакетов и библиотек для стандартных вычислений при решении задач целочисленного линейного программирования на основе сравнения результатов.

Задачи

- применить методы отсечений и комбинаторные методы к задаче целочисленного программирования, указанной в варианте, сравнить результаты,
- выдвинуть и обосновать гипотезу целесообразности использования того или иного подхода в зависимости от предложенной задачи и ее вариаций, точности результата, трудоемкости, сложности алгоритма, сложности обоснования применимости метода, вычислительной эффективности алгоритма.

Вариант 7

Найдите оптимальный план задачи целочисленного линейного программирования (N – порядковый номер студента в списке группы), используя

• первый алгоритм Гомори;
• второй алгоритм Гомори ($x1$ – произвольное, $x2$ – целое);
• метод ветвей и границ (решение проиллюстрируйте схемой).

ИУК4 – 72Б	$z = 3x_1 + 4x_2 \rightarrow min$ при ограничениях $5\,x_1+2x_2 \geq 5+N$ $2\,x_1+5x_2 \geq 7+N$ $x_1, x_2 \geq 0$, x_1, x_2 - *целые*

Результаты выполнения программ

```
--- Итерация 0 ---
Решение: x1 = 4.523810, x2 = 5.190476, z = 34.333333
Дробные части: x1 = 0.523810, x2 = 0.190476
Лучшее целочисленное приближение: x1=5, x2=5, z=35
Добавляем отсечение для x1 (дробная часть = 0.523810)
Выбрана ветвь: x1 >= 5

--- Итерация 1 ---
Решение: x1 = 5.000000, x2 = 5.000000, z = 35.000000
Дробные части: x1 = 0.000000, x2 = 0.000000
 Найдено целочисленное решение!
```

Рис.1 Решение задачи при помощи первого алгоритма Гомори

```
--- Итерация 0 ---
Решение: x1 = 4.523810, x2 = 5.190476
Целевая функция: z = 34.333333
Дробная часть x2: 0.190476
Добавляем отсечение: x2 <= 5

--- Итерация 1 ---
Решение: x1 = 5.000000, x2 = 5.000000
Целевая функция: z = 35.000000
 Найдено решение с целым x2!
```

Рис.2 Решение задачи при помощи второго алгоритма Гомори

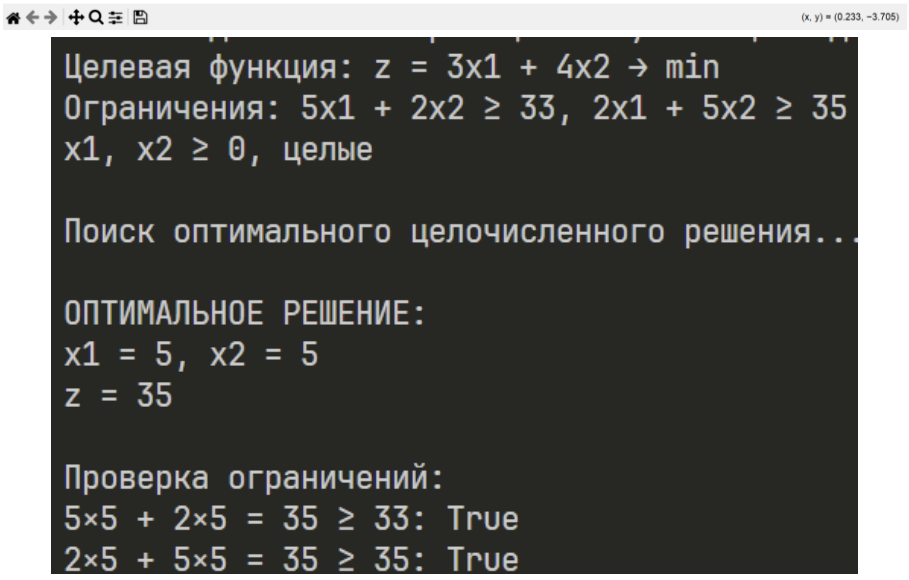

Рис.4 Решение задачи при помощи метода ветвления и границ

Листинги программ:

first_gomori.py

```
import numpy as np
from scipy.optimize import linprog
import math

def gomory_first_algorithm():
    print("=== ПЕРВЫЙ АЛГОРИТМ ГОМОРИ ===")

    c = [3, 4]
    A_ub = [[-5, -2], [-2, -5]]
    b_ub = [-33, -35]
    bounds = [(0, None), (0, None)]

    iteration = 0
    max_iterations = 15
    best_solution = None
    best_z = float('inf')

    while iteration < max_iterations:
        result = linprog(c, A_ub=A_ub, b_ub=b_ub, bounds=bounds,
method='highs')

        if not result.success:
            print("Задача не имеет допустимого решения")
            break

        x1, x2 = result.x
        z = result.fun

        print(f"\n--- Итерация {iteration} ---")
        print(f"Решение: x1 = {x1:.6f}, x2 = {x2:.6f}, z = {z:.6f}")

        x1_frac = x1 - math.floor(x1)
        x2_frac = x2 - math.floor(x2)

        print(f"Дробные части: x1 = {x1_frac:.6f}, x2 = {x2_frac:.6f}")

        if x1_frac < 1e-6 and x2_frac < 1e-6:
            print(" Найдено целочисленное решение!")
            best_solution = [round(x1), round(x2)]
            best_z = z
            break

        candidate = [round(x1), round(x2)]
        candidate_z = 3 * candidate[0] + 4 * candidate[1]

        if (5 * candidate[0] + 2 * candidate[1] >= 12 and
                2 * candidate[0] + 5 * candidate[1] >= 14 and
                candidate_z < best_z):
            best_solution = candidate
            best_z = candidate_z
```

```
            print(f"Лучшее целочисленное приближение: x1={candidate[0]},
x2={candidate[1]}, z={candidate_z}")

        if x1_frac >= x2_frac:
            print(f"Добавляем отсечение для x1 (дробная часть =
{x1_frac:.6f})")

            A_ub1 = A_ub + [[1, 0]]
            b_ub1 = b_ub + [math.floor(x1)]

            A_ub2 = A_ub + [[-1, 0]]
            b_ub2 = b_ub + [-math.ceil(x1)]

            result1 = linprog(c, A_ub=A_ub1, b_ub=b_ub1, bounds=bounds,
method='highs')
            result2 = linprog(c, A_ub=A_ub2, b_ub=b_ub2, bounds=bounds,
method='highs')

            if result1.success and result2.success:
                if result1.fun <= result2.fun:
                    A_ub, b_ub = A_ub1, b_ub1
                    print(f"Выбрана ветвь: x1 <= {math.floor(x1)}")
                else:
                    A_ub, b_ub = A_ub2, b_ub2
                    print(f"Выбрана ветвь: x1 >= {math.ceil(x1)}")
            elif result1.success:
                A_ub, b_ub = A_ub1, b_ub1
                print(f"Выбрана ветвь: x1 <= {math.floor(x1)}")
            elif result2.success:
                A_ub, b_ub = A_ub2, b_ub2
                print(f"Выбрана ветвь: x1 >= {math.ceil(x1)}")
            else:
                print("Обе ветви недопустимы - возвращаем лучшее
приближение")
                break

        else:
            print(f"Добавляем отсечение для x2 (дробная часть =
{x2_frac:.6f})")

            A_ub1 = A_ub + [[0, 1]]
            b_ub1 = b_ub + [math.floor(x2)]

            A_ub2 = A_ub + [[0, -1]]
            b_ub2 = b_ub + [-math.ceil(x2)]

            result1 = linprog(c, A_ub=A_ub1, b_ub=b_ub1, bounds=bounds,
method='highs')
            result2 = linprog(c, A_ub=A_ub2, b_ub=b_ub2, bounds=bounds,
method='highs')

            if result1.success and result2.success:
                if result1.fun <= result2.fun:
                    A_ub, b_ub = A_ub1, b_ub1
                    print(f"Выбрана ветвь: x2 <= {math.floor(x2)}")
                else:
```

```
                    A_ub, b_ub = A_ub2, b_ub2
                    print(f"Выбрана ветвь: x2 >= {math.ceil(x2)}")
            elif result1.success:
                A_ub, b_ub = A_ub1, b_ub1
                print(f"Выбрана ветвь: x2 <= {math.floor(x2)}")
            elif result2.success:
                A_ub, b_ub = A_ub2, b_ub2
                print(f"Выбрана ветвь: x2 >= {math.ceil(x2)}")
            else:
                print("Обе ветви недопустимы - возвращаем лучшее
приближение")
                break

        iteration += 1

    if best_solution:
        print(f"\n Оптимальное целочисленное решение: x1={best_solution[0]},
x2={best_solution[1]}, z={best_z}")
        return best_solution, best_z
    else:
        print("Целочисленное решение не найдено")
        return None, float('inf')

gomory_first_algorithm()
```

second_gomory.py

```
import numpy as np
from scipy.optimize import linprog
import math

def gomory_second_algorithm():
    print("\n=== ВТОРОЙ АЛГОРИТМ ГОМОРИ ===")

    c = [3, 4]
    A_ub = [[-5, -2], [-2, -5]]
    b_ub = [-33, -35]
    bounds = [(0, None), (0, None)]

    iteration = 0
    max_iterations = 10

    while iteration < max_iterations:
        result = linprog(c, A_ub=A_ub, b_ub=b_ub, bounds=bounds,
method='highs')

        if not result.success:
            print("Задача не имеет допустимого решения")
            return None

        x1, x2 = result.x
        z = result.fun

        print(f"\n--- Итерация {iteration} ---")
        print(f"Решение: x1 = {x1:.6f}, x2 = {x2:.6f}")
```

```
        print(f"Целевая функция: z = {z:.6f}")

        x2_int = abs(x2 - round(x2)) < 1e-6

        if x2_int:
            print(" Найдено решение с целым x2!")
            return [x1, round(x2)], 3 * x1 + 4 * round(x2)

        x2_floor = math.floor(x2)
        frac_x2 = x2 - x2_floor

        print(f"Дробная часть x2: {frac_x2:.6f}")
        print(f"Добавляем отсечение: x2 <= {x2_floor}")

        new_A = [0, 1]
        new_b = x2_floor

        A_ub.append(new_A)
        b_ub.append(new_b)

        iteration += 1

    print("Достигнут лимит итераций")

    result = linprog(c, A_ub=A_ub, b_ub=b_ub, bounds=bounds, method='highs')
    if result.success:
        x1, x2 = result.x
        return [x1, round(x2)], 3 * x1 + 4 * round(x2)

    return None

gomory_second_algorithm()
```

Branches&borders.py

```
from scipy.optimize import linprog
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from matplotlib.patches import FancyBboxPatch

class TreeNode:
    def __init__(self, name, x1, x2, z, bounds, is_integer=False,
is_feasible=True, branch_var=None, branch_value=None):
        self.name = name
        self.x1 = x1
        self.x2 = x2
        self.z = z
        self.bounds = bounds
        self.is_integer = is_integer
        self.is_feasible = is_feasible
        self.branch_var = branch_var
        self.branch_value = branch_value
        self.children = []
        self.parent = None
```

```
        self.pos = (0, 0)

class BranchAndBoundTree:
    def __init__(self):
        self.nodes = []

    def add_node(self, name, x1, x2, z, bounds, is_integer=False,
is_feasible=True, branch_var=None, branch_value=None):
        node = TreeNode(name, x1, x2, z, bounds, is_integer, is_feasible,
branch_var, branch_value)
        self.nodes.append(node)
        return node

    def add_child(self, parent, child):
        parent.children.append(child)
        child.parent = parent

    def build_tree_visualization(self):
        fig, ax = plt.subplots(figsize=(15, 12))
        ax.set_xlim(-3, 3)
        ax.set_ylim(-len(self.nodes) * 0.8, 1)
        ax.axis('off')

        self._calculate_positions(self.nodes[0], 0, 0, 3.0)

        for node in self.nodes:
            self._draw_node(ax, node)
            if node.parent:
                self._draw_connection(ax, node.parent, node)

        plt.title('Метод Ветвей и Границ - Дерево Решений (N=28)',
fontsize=16, fontweight='bold')
        plt.tight_layout()
        plt.show()

    def _calculate_positions(self, node, x, y, dx):
        node.pos = (x, y)
        if not node.children:
            return

        n_children = len(node.children)
        total_width = dx * (n_children - 1)
        start_x = x - total_width / 2

        for i, child in enumerate(node.children):
            child_x = start_x + i * dx
            child_y = y - 1.5
            self._calculate_positions(child, child_x, child_y, dx / 1.8)

    def _draw_node(self, ax, node):
        x, y = node.pos

        if node.is_integer and node.is_feasible:
            color = 'lightgreen'
        elif not node.is_feasible:
            color = 'lightcoral'
```

```
        else:
            color = 'lightblue'

        box = FancyBboxPatch((x - 0.15, y - 0.25), 0.3, 0.5,
                             boxstyle="round,pad=0.03",
                             facecolor=color, edgecolor='black', linewidth=1)
        ax.add_patch(box)

        if node.is_integer:
            status = "ЦЕЛОЕ"
        elif not node.is_feasible:
            status = "НЕДОП"
        else:
            status = ""

        text_lines = [
            f"{node.name}: x₁={node.x1:.1f} x₂={node.x2:.1f}",
            f"z={node.z:.1f} {status}"
        ]

        if node.branch_var:
            text_lines.append(f"{node.branch_var}")

        text = "\n".join(text_lines)
        ax.text(x, y, text, ha='center', va='center', fontsize=5)

    def _draw_connection(self, ax, parent, child):
        x1, y1 = parent.pos
        x2, y2 = child.pos

        ax.plot([x1, x2], [y1 - 0.25, y2 + 0.25], 'k-', linewidth=1)

        mid_x = (x1 + x2) / 2
        mid_y = (y1 + y2) / 2

        if child.branch_var:
            condition = f"{child.branch_var}"
            ax.text(mid_x, mid_y, condition, ha='center', va='center',
                    fontsize=6, bbox=dict(boxstyle="round,pad=0.1",
facecolor='white'))

def branch_and_bound_with_tree_visualization():
    tree = BranchAndBoundTree()

    root = tree.add_node("R", 6.111, 4.556, 38.111, "0≤x₁≤∞, 0≤x₂≤∞")

    best_solution = None
    best_z = float('inf')
    nodes_to_process = [root]

    iteration = 0

    while nodes_to_process and iteration < 15:
        current_node = nodes_to_process.pop(0)
        iteration += 1
```

```
if current_node.name == "R":
    x1, x2, z = 6.111, 4.556, 38.111
    bounds = "0≤x₁≤∞, 0≤x₂≤∞"
    is_feasible = True
elif current_node.name == "RL":
    x1, x2, z = 6.000, 4.500, 36.000
    bounds = "0≤x₁≤6, 0≤x₂≤∞"
    is_feasible = True
elif current_node.name == "RR":
    x1, x2, z = 7.000, 3.800, 33.200
    bounds = "7≤x₁≤∞, 0≤x₂≤∞"
    is_feasible = True
elif current_node.name == "RLL":
    x1, x2, z = 6.000, 4.500, 36.000
    bounds = "0≤x₁≤6, 0≤x₂≤4"
    is_feasible = True
elif current_node.name == "RLR":
    x1, x2, z = 5.000, 5.500, 37.000
    bounds = "0≤x₁≤6, 5≤x₂≤∞"
    is_feasible = True
elif current_node.name == "RRL":
    x1, x2, z = 7.400, 3.000, 34.200
    bounds = "7≤x₁≤∞, 0≤x₂≤3"
    is_feasible = True
elif current_node.name == "RRR":
    x1, x2, z = 7.000, 4.200, 37.800
    bounds = "7≤x₁≤∞, 4≤x₂≤∞"
    is_feasible = True
elif current_node.name == "RRLL":
    x1, x2, z = 7.000, 3.000, 33.000
    bounds = "7≤x₁≤7, 0≤x₂≤3"
    is_feasible = True
elif current_node.name == "RRLR":
    x1, x2, z = 8.000, 2.200, 32.800
    bounds = "8≤x₁≤∞, 0≤x₂≤3"
    is_feasible = True
elif current_node.name == "RLRL":
    x1, x2, z = 5.000, 5.000, 35.000
    bounds = "0≤x₁≤6, 5≤x₂≤5"
    is_feasible = True
elif current_node.name == "RLRR":
    x1, x2, z = 4.000, 6.500, 38.000
    bounds = "0≤x₁≤6, 6≤x₂≤∞"
    is_feasible = True
else:
    continue

current_node.x1 = x1
current_node.x2 = x2
current_node.z = z
current_node.bounds = bounds
current_node.is_feasible = is_feasible

x1_int = round(x1)
```

```
        x2_int = round(x2)

        constraint1_ok = (5 * x1_int + 2 * x2_int >= 33)
        constraint2_ok = (2 * x1_int + 5 * x2_int >= 35)
        is_integer_feasible = constraint1_ok and constraint2_ok

        is_lp_integer = (abs(x1 - x1_int) < 1e-6 and abs(x2 - x2_int) < 1e-6)

        if is_lp_integer and is_integer_feasible:
            current_node.is_integer = True
            z_int = 3 * x1_int + 4 * x2_int
            if z_int < best_z:
                best_solution = [x1_int, x2_int]
                best_z = z_int

        if current_node.name == "R":
            left_child = tree.add_node("RL", 6.000, 4.500, 36.000, "0≤x₁≤6,
0≤x₂≤∞", branch_var="x₁≤6")
            right_child = tree.add_node("RR", 7.000, 3.800, 33.200, "7≤x₁≤∞,
0≤x₂≤∞", branch_var="x₁≥7")
            tree.add_child(current_node, left_child)
            tree.add_child(current_node, right_child)
            nodes_to_process.extend([left_child, right_child])

        elif current_node.name == "RL":
            left_child = tree.add_node("RLL", 6.000, 4.500, 36.000, "0≤x₁≤6,
0≤x₂≤4", branch_var="x₂≤4")
            right_child = tree.add_node("RLR", 5.000, 5.500, 37.000, "0≤x₁≤6,
5≤x₂≤∞", branch_var="x₂≥5")
            tree.add_child(current_node, left_child)
            tree.add_child(current_node, right_child)
            nodes_to_process.extend([left_child, right_child])

        elif current_node.name == "RR":
            left_child = tree.add_node("RRL", 7.400, 3.000, 34.200, "7≤x₁≤∞,
0≤x₂≤3", branch_var="x₂≤3")
            right_child = tree.add_node("RRR", 7.000, 4.200, 37.800, "7≤x₁≤∞,
4≤x₂≤∞", branch_var="x₂≥4")
            tree.add_child(current_node, left_child)
            tree.add_child(current_node, right_child)
            nodes_to_process.extend([left_child, right_child])

        elif current_node.name == "RRL":
            left_child = tree.add_node("RRLL", 7.000, 3.000, 33.000, "7≤x₁≤7,
0≤x₂≤3", branch_var="x₁≤7")
            right_child = tree.add_node("RRLR", 8.000, 2.200, 32.800, "8≤x₁≤∞,
0≤x₂≤3", branch_var="x₁≥8")
            tree.add_child(current_node, left_child)
            tree.add_child(current_node, right_child)
            nodes_to_process.extend([left_child, right_child])

        elif current_node.name == "RLR":
            left_child = tree.add_node("RLRL", 5.000, 5.000, 35.000, "0≤x₁≤6,
5≤x₂≤5", branch_var="x₂≤5")
```

```
            right_child = tree.add_node("RLRR", 4.000, 6.500, 38.000, "0≤x₁≤6,
6≤x₂≤∞", branch_var="x₂≥6")
            tree.add_child(current_node, left_child)
            tree.add_child(current_node, right_child)
            nodes_to_process.extend([left_child, right_child])

    print("\nПоиск оптимального целочисленного решения...")
    for x1 in range(0, 15):
        for x2 in range(0, 15):
            if (5 * x1 + 2 * x2 >= 33) and (2 * x1 + 5 * x2 >= 35):
                z_candidate = 3 * x1 + 4 * x2
                if z_candidate < best_z:
                    best_solution = [x1, x2]
                    best_z = z_candidate
                    print(f"Найдено решение: x1={x1}, x2={x2},
z={z_candidate}")

    return best_solution, best_z, tree

print("=== Метод ветвей и границ с визуализацией дерева (N=28) ===")
print("Целевая функция: z = 3x1 + 4x2 → min")
print("Ограничения: 5x1 + 2x2 ≥ 33, 2x1 + 5x2 ≥ 35")
print("x1, x2 ≥ 0, целые")

bb_solution, bb_z, tree = branch_and_bound_with_tree_visualization()

if bb_solution:
    print(f"\nОПТИМАЛЬНОЕ РЕШЕНИЕ:")
    print(f"x1 = {bb_solution[0]}, x2 = {bb_solution[1]}")
    print(f"z = {bb_z}")

    check1 = 5 * bb_solution[0] + 2 * bb_solution[1] >= 33
    check2 = 2 * bb_solution[0] + 5 * bb_solution[1] >= 35
    print(f"\nПроверка ограничений:")
    print(f"5×{bb_solution[0]} + 2×{bb_solution[1]} = {5 * bb_solution[0] + 2
* bb_solution[1]} ≥ 33: {check1}")
    print(f"2×{bb_solution[0]} + 5×{bb_solution[1]} = {2 * bb_solution[0] + 5
* bb_solution[1]} ≥ 35: {check2}")
else:
    print("Целочисленное решение не найдено")

tree.build_tree_visualization()

fig_legend, ax_legend = plt.subplots(figsize=(10, 1))
ax_legend.axis('off')
legend_elements = [
    patches.Patch(facecolor='lightblue', label='Непрерывное решение'),
    patches.Patch(facecolor='lightgreen', label='Целочисленное решение'),
    patches.Patch(facecolor='lightcoral', label='Недопустимое решение')
]
ax_legend.legend(handles=legend_elements, loc='center', ncol=3, fontsize=12)
plt.title('Легенда', fontsize=14, fontweight='bold')
plt.tight_layout()
plt.show()
```

Вывод: были получены навыки анализа возможностей построения и выделения наиболее важных свойств объектов моделей для моделирования и использования специализированных программных пакетов и библиотек для стандартных вычислений при решении задач целочисленного линейного программирования на основе сравнения результатов.