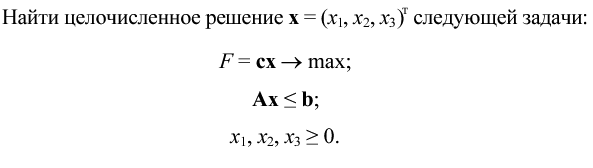
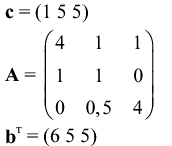
**Целочисленное линейное программирование. Метод ветвей и границ**

**Цель:** изучить постановку задачи целочисленного линейного программирования (ЦЛП); получить навыки решения задачи ЦЛП методом ветвей и границ (МВГ).

**Вариант №8**





**Результаты работы:**

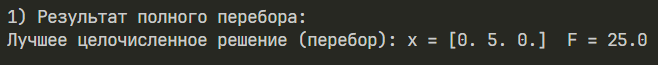


Рисунок 1 Результат перебора

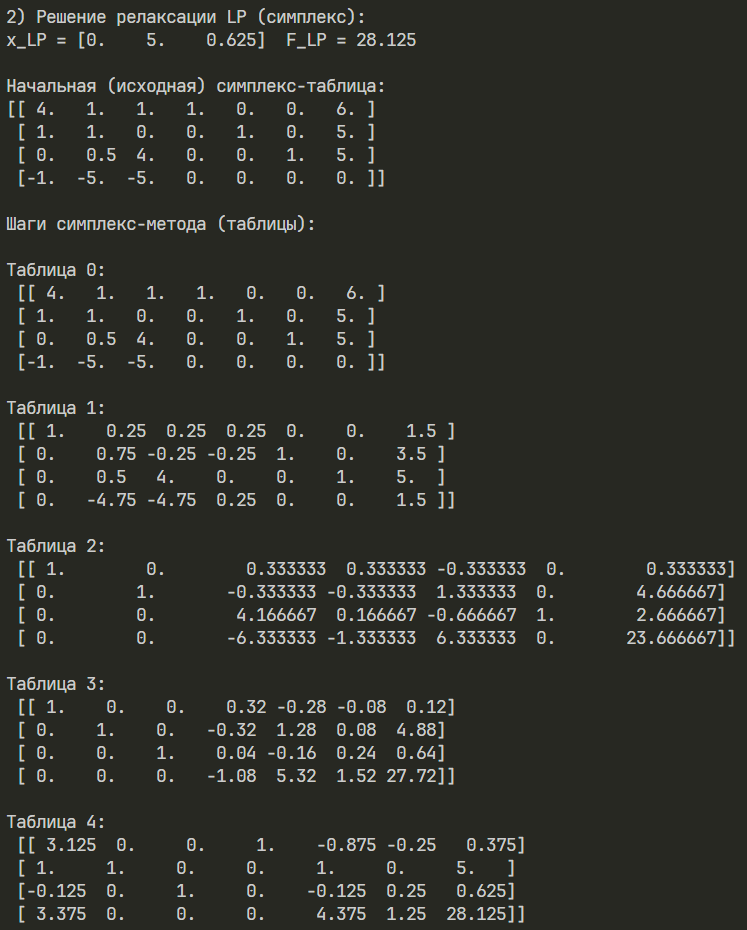


Рисунок 2 Симплекс

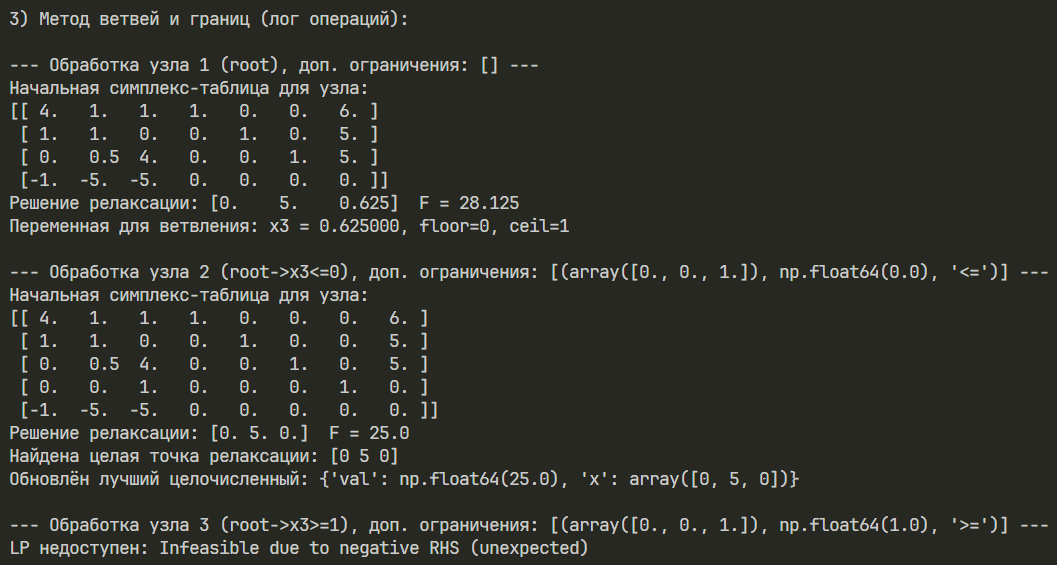


Рисунок 3 Метод ветвей и границ

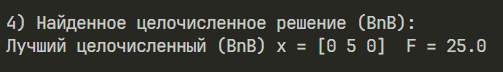


Рисунок 4 Целочисленное решение

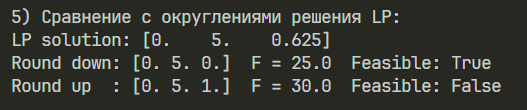


Рисунок 5 Сравнение с округлениями решения LP

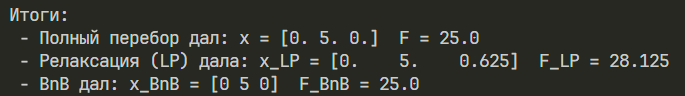


Рисунок 6 Итог

**Листинг программы:**

import numpy as np

c = np.array([1.0, 5.0, 5.0])

A = np.array([[4.0, 1.0, 1.0],

              [1.0, 1.0, 0.0],

              [0.0, 0.5, 4.0]])

b = np.array([6.0, 5.0, 5.0])

n = c.size

m = b.size

*def* compute\_variable\_upper\_bounds(*A*, *b*):

    bounds = []

    for j in range(*A*.shape[1]):

        vals = []

        for i in range(*A*.shape[0]):

            if *A*[i,j] > 1e-12:

                vals.append(*b*[i] / *A*[i,j])

        if vals:

            bounds.append(int(np.floor(min(vals))))

        else:

            bounds.append(0)

    return bounds

*class* SimplexSolver:

*def* \_\_init\_\_(*self*, *A*, *b*, *c*):

*self*.A = *A*.astype(float)

*self*.b = *b*.astype(float)

*self*.c = *c*.astype(float)

*self*.m, *self*.n = *A*.shape[0], *A*.shape[1]

*self*.table = None

*self*.basic = None

*self*.nonbasic = None

*self*.tableaus = []

*self*.\_build\_initial\_tableau()

*def* \_build\_initial\_tableau(*self*):

        m, n = *self*.m, *self*.n

        T = np.zeros((m+1, n+m+1))

        T[0:m, 0:n] = *self*.A

        T[0:m, n:n+m] = np.eye(m)

        T[0:m, -1] = *self*.b

        T[m, 0:n] = -*self*.c

        T[m, -1] = 0.0

*self*.table = T

*self*.basic = list(range(n, n+m))

*self*.nonbasic = list(range(0, n))

*self*.tableaus.append(*self*.table.copy())

*def* \_is\_optimal(*self*):

        return np.all(*self*.table[-1, :-1] >= -1e-9)

*def* \_select\_pivot(*self*):

        reduced = *self*.table[-1, :-1]

        for j, val in enumerate(reduced):

            if val < -1e-9:

                pivot\_col = j

                break

        else:

            return None, None

        col = *self*.table[0:*self*.m, pivot\_col]

        rhs = *self*.table[0:*self*.m, -1]

        ratios = []

        for i in range(*self*.m):

            if col[i] > 1e-12:

                ratios.append((rhs[i] / col[i], i))

        if not ratios:

            return pivot\_col, None

        ratios.sort(*key*=*lambda* *x*: (*x*[0], *x*[1]))

        pivot\_row = ratios[0][1]

        return pivot\_col, pivot\_row

*def* \_pivot(*self*, *pivot\_row*, *pivot\_col*):

        T = *self*.table

        piv = T[*pivot\_row*, *pivot\_col*]

        T[*pivot\_row*, :] = T[*pivot\_row*, :] / piv

        for i in range(T.shape[0]):

            if i != *pivot\_row*:

                factor = T[i, *pivot\_col*]

                T[i, :] = T[i, :] - factor \* T[*pivot\_row*, :]

*self*.basic[*pivot\_row*] = *pivot\_col*

*self*.tableaus.append(*self*.table.copy())

*def* solve(*self*, *max\_iters*=200):

        it = 0

        while (not *self*.\_is\_optimal()) and it < *max\_iters*:

            pivot\_col, pivot\_row = *self*.\_select\_pivot()

            if pivot\_col is None:

                break

            if pivot\_row is None:

                raise Exception("Unbounded LP.")

*self*.\_pivot(pivot\_row, pivot\_col)

            it += 1

        sol = np.zeros(*self*.n + *self*.m)

        for i, colidx in enumerate(*self*.basic):

            if colidx < *self*.n + *self*.m:

                sol[colidx] = *self*.table[i, -1]

        x = sol[0:*self*.n]

        z = *self*.table[-1, -1]

        return x, z, *self*.tableaus

bounds = compute\_variable\_upper\_bounds(A, b)

print("Оценки верхних границ для перебора:", bounds)

brute\_best\_val = -1e9

brute\_best\_x = None

for x1 in range(bounds[0] + 1):

    for x2 in range(bounds[1] + 1):

        for x3 in range(bounds[2] + 1):

            x = np.array([x1, x2, x3], *dtype*=float)

            if np.all(A.dot(x) <= b + 1e-9):

                val = c.dot(x)

                if val > brute\_best\_val:

                    brute\_best\_val = val

                    brute\_best\_x = x.copy()

print("\n1) Результат полного перебора:")

print("Лучшее целочисленное решение (перебор): x =", brute\_best\_x, " F =", brute\_best\_val)

solver = SimplexSolver(A, b, c)

x\_lp, z\_lp, tableaus\_lp = solver.solve()

print("\n2) Решение релаксации LP (симплекс):")

print("x\_LP =", np.round(x\_lp,6), " F\_LP =", round(z\_lp,6))

print("\nНачальная (исходная) симплекс-таблица:")

np.set\_printoptions(*precision*=6, *suppress*=True)

print(tableaus\_lp[0])

print("\nШаги симплекс-метода (таблицы):")

for i, T in enumerate(tableaus\_lp):

    print(*f*"\nТаблица {i}:\n", T)

*class* BnBNode:

*def* \_\_init\_\_(*self*, *extra\_constraints*=None, *name*="root"):

*self*.extra\_constraints = *extra\_constraints* or []

*self*.name = *name*

*def* solve\_lp\_with\_extra\_constraints(*A*, *b*, *c*, *extra\_constraints*):

    A\_ext = *A*.copy()

    b\_ext = *b*.copy()

    for row, rhs, sense in *extra\_constraints*:

        if sense == '<=':

            A\_ext = np.vstack([A\_ext, np.array(row, *dtype*=float)])

            b\_ext = np.append(b\_ext, float(rhs))

        elif sense == '>=':

            A\_ext = np.vstack([A\_ext, -np.array(row, *dtype*=float)])

            b\_ext = np.append(b\_ext, -float(rhs))

        else:

            raise ValueError("sense must be '<=' or '>='")

    solver = SimplexSolver(A\_ext, b\_ext, *c*)

    try:

        x\_lp, z\_lp, tableaus = solver.solve()

    except Exception as e:

        return None, None, None, str(e)

    if np.any(b\_ext < -1e-9):

        return None, None, None, "Infeasible due to negative RHS"

    return x\_lp, z\_lp, tableaus, None

best\_incumbent = {'val': -1e9, 'x': None}

stack = [BnBNode(*extra\_constraints*=[], *name*="root")]

print("\n3) Метод ветвей и границ (лог операций):")

node\_counter = 0

while stack:

    node = stack.pop()

    node\_counter += 1

    print(*f*"\n--- Обработка узла {node\_counter} ({node.name}), доп. ограничения: {node.extra\_constraints} ---")

    x\_relax, z\_relax, tableaus\_relax, err = solve\_lp\_with\_extra\_constraints(A, b, c, node.extra\_constraints)

    if err:

        print("LP недоступен:", err)

        continue

    print("Начальная симплекс-таблица для узла:")

    print(tableaus\_relax[0])

    print("Решение релаксации:", np.round(x\_relax,8), " F =", z\_relax)

    if z\_relax <= best\_incumbent['val'] + 1e-9:

        print("Отсечено: оценка <= лучший целочисленный.")

        continue

    frac\_indices = [j for j in range(len(x\_relax)) if abs(x\_relax[j] - round(x\_relax[j])) > 1e-9]

    if not frac\_indices:

        print("Найдена целая точка релаксации:", np.round(x\_relax).astype(int))

        if z\_relax > best\_incumbent['val'] + 1e-9:

            best\_incumbent['val'] = z\_relax

            best\_incumbent['x'] = np.round(x\_relax).astype(int)

            print("Обновлён лучший целочисленный:", best\_incumbent)

        continue

    frac\_parts = [(j, abs(x\_relax[j] - np.floor(x\_relax[j]))) for j in frac\_indices]

    frac\_parts.sort(*key*=*lambda* *t*: -*t*[1])

    j\_branch = frac\_parts[0][0]

    xj = x\_relax[j\_branch]

    lower = np.floor(xj)

    upper = np.ceil(xj)

    print(*f*"Переменная для ветвления: x{j\_branch+1} = {xj*:.6f*}, floor={int(lower)}, ceil={int(upper)}")

    left\_constraints = node.extra\_constraints + [(np.eye(1, n, j\_branch).flatten(), lower, '<=')]

    right\_constraints = node.extra\_constraints + [(np.eye(1, n, j\_branch).flatten(), upper, '>=')]

    stack.append(BnBNode(*extra\_constraints*=right\_constraints, *name*=node.name + *f*"->x{j\_branch+1}>={int(upper)}"))

    stack.append(BnBNode(*extra\_constraints*=left\_constraints, *name*=node.name + *f*"->x{j\_branch+1}<={int(lower)}"))

print("\n4) Найденное целочисленное решение (BnB):")

print("Лучший целочисленный (BnB) x =", best\_incumbent['x'], " F =", best\_incumbent['val'])

print("\n5) Сравнение с округлениями решения LP:")

x\_lp\_rounded\_down = np.floor(x\_lp)

x\_lp\_rounded\_up = np.ceil(x\_lp)

*def* is\_feasible(*x*):

    return np.all(A.dot(*x*) <= b + 1e-9) and np.all(*x* >= -1e-9)

print("LP solution:", x\_lp)

print("Round down:", x\_lp\_rounded\_down, " F =", c.dot(x\_lp\_rounded\_down), " Feasible:", is\_feasible(x\_lp\_rounded\_down))

print("Round up  :", x\_lp\_rounded\_up, " F =", c.dot(x\_lp\_rounded\_up), " Feasible:", is\_feasible(x\_lp\_rounded\_up))

print("\nИтоги:")

print(" - Полный перебор дал: x =", brute\_best\_x, " F =", brute\_best\_val)

print(" - Релаксация (LP) дала: x\_LP =", np.round(x\_lp,6), " F\_LP =", round(z\_lp,6))

print(" - BnB дал: x\_BnB =", best\_incumbent['x'], " F\_BnB =", best\_incumbent['val'])

**Вывод:** в ходе выполнения лабораторной работы была поставлена и решена задача целочисленного линейного программирования по МВГ.