МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Тольяттинский государственный университет»
Институт математики, физики и информационных технологий
(наименование института полностью)

Кафедра «Прикладная математика и информатика»
(наименование)

01.04.02 Прикладная математика и информатика
(код и наименование направления подготовки)

Математическое моделирование
(направленность (профиль))

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ)

на тему Исследование алгоритмов решения целочисленных и частично целочисленных задач линейного программирования

Студент Е.А. Цель
(И.О. Фамилия) (личная подпись)

Научный Руководитель к. ф-м. н., доцент кафедры "ПМИ", О.В. Лелонд
(ученая степень, звание, И.О. Фамилия)

Тольятти 2021

Содержание

Введение

Линейное программирование (ЛП) посвящено методам решения экстремальных задач. Данные задачи основаны на нахождении минимального и максимального значений функции на множествах -мерного векторного пространства.

Актуальность. Задачи целочисленного и частично целочисленного линейного программирования на сегодняшнее время применяются в различных сферах, таких как производственное планирование, телекоммуникационные сети, сотовая сеть.

Объект исследования – задачи целочисленного и частично целочисленного линейного программирования.

Предмет исследования – методы решения задач целочисленного и частично целочисленного линейного программирования.

Цель исследования – анализ методов решения задач целочисленного и частично целочисленного линейного программирования и реализация рассмотренных алгоритмов и их модификаций.

Для достижения цели будут решены следующие задачи:

- исследовать методы решения задач целочисленного и частично целочисленного линейного программирования;
- создать программную реализацию алгоритмов рассмотренных методов на языке программирования Python;
- провести сравнительный анализ результатов работы программы;
- провести сравнительный анализ на модели приближенной к реальным данным.

Магистерская диссертационная работа включает в себя введение, четыре раздела, заключение, список использованной литературы и два приложения. Данная диссертация представлена на 101 странице, содержит 35 рисунков, 27 таблиц, 62 формулы, 31 использованных литературных источников.

1 Обзор существующих методов решения задачи линейного программирования

1.1 Общая характеристика задачи целочисленного и частично целочисленного линейного программирования

Задачи целочисленного линейного программирования (ЗЦЛП) – это математические задачи оптимизации, в которых целевая функция, её ограничения и переменные линейны. Целочисленное программирование является NP-полной задачей [1]. Известную задачу коммивояжера также можно свести к задаче целочисленного линейного программирования [3].

Если ограничения в целевой функции накладываются только на часть переменных, то такая задача называется частично целочисленной [29].

В задаче целочисленного или частично целочисленного линейного программирования некоторая переменная может принимать только значения 0 или 1, в этом случае такая переменная называется булевой [28]. Если в задаче все переменные булевы, то она называется задачей булева линейного программирования. Частично целочисленное линейное программирование построено на основе метода Данцига [2].

Многие практические задачи можно свести к задачам целочисленного или частично целочисленного линейного программирования, например, задача о рюкзаке, задача коммивояжера, задача с фиксированными доплатами, задача о выполнимости КНФ, задача о «раскрое» и другие [4].

1.1.1 Математическая модель задачи целочисленного и частично целочисленного линейного программирования

Математическая модель задачи целочисленного и частично целочисленного линейного программирования имеет вид:

$$Z = \sum_{j=1}^{m} c_j x_j \to max. \quad (1)$$

при ограничениях:

$$\begin{cases} \sum_{j=1}^{m} a_{ij} x_j \le b_i, i = \overline{1,m}, & (2) \\ x_j \ge 0, j = \overline{1,n}, & (3) \\ x_j \in \{0, N\}, j = \overline{1,k}; k \le n. & (4) \end{cases},$$

где N – множество натуральных чисел;

n – общее число всех переменных.

Существует два варианта зависимости количества целочисленных переменных от общего числа переменных:

- если $k = n$, то в решении все переменные должны быть целыми;
- если $k < n$, то в решении только k переменных должны быть целыми, остальные могут принимать любые значение в граничных условиях.

Для нахождения решения задачи целочисленного или частично целочисленного линейного программирования необходимо, чтобы все переменные имели верхнюю границу.

1.2 Методы решения задачи целочисленного и частично целочисленного линейного программирования

Существует две основных категории алгоритмов для задач целочисленного и частично целочисленного линейного программирования [5].

Точные алгоритмы, которые гарантируют нахождение оптимального решения, но могут иметь экспоненциальное число итераций;

Эвристические алгоритмы, обеспечивающие нахождение неоптимального решения [10]. Хотя время выполнения не обязано быть полиномиальным, эмпирические данные свидетельствуют о том, что некоторые алгоритмы способны быстро найти приближенно оптимальное решение [27].

Приведем примеры точечных методов:

– метод ветвей и границ;
– метод Гомори (метод секущих плоскостей);
– методы динамического программирования.

Примерами эвристических методов являются:

– восхождение по выпуклой поверхности;
– алгоритм имитации отжига;
– муравьиный алгоритм;
– пассивная поисковая оптимизация;
– нейронная сеть Хопфилда;
– k-opt эвристика (задача коммивояжера);
– поиск с запретами.

Далее будут рассмотрены некоторые методы из данных списков.

1.2.1 Метод ветвей и границ

Впервые метод ветвей и границ для решения задач целочисленного линейного программирования был предложен Лендом и Дойгом в 1960 году. В основе метода лежит идея разбиения множества допустимых решений на подмножества. Метод ветвей и границ относится к комбинаторным методам решения задач целочисленного линейного программирования, также данный метод применим и для частично целочисленных задач [6].

Суть метода ветвей и границ состоит в том, чтобы перебрать последовательно все варианты решения и отобрать лишь те, которые

оказались по каким-либо признакам лучшими или перспективными, при этом нужно отбросить все остальные.

В ходе реализации метода ветвей и границ формируется дерево решений. Область допустимых значений разделяется на множества, которые между собой не пересекаются, и задачи на данных множествах решаются с той же целевой функцией, но уже без учета условий целочисленности. Если в результате решения задачи оптимальное решения не является целочисленным, то множество снова разбивается на два, так до тех пор, пока не будет найдено оптимальное целочисленное решение [26].

Если в процессе исследования задачи на поиск минимума получается несколько оптимальных целочисленных решений, то учитываются только те, которые соответствуют убывающему значению целевой функции [8].

Границы, в которых находятся неизвестные значения, записываются в виде:

$$\alpha_j \leq x_j \leq \beta_j. \tag{5}$$

В начале метода ветвей и границ задача решается, например, симплекс-методом, результатом которого является оптимальное (возможно дробное) значение координаты x_k [9].

Для разветвления в методе ветвей и границ введем два вида ограничений. Первая ветвь будет определяться неравенством:

$$\alpha_k \leq x_k \leq [x_k]. \tag{6}$$

В другой ветви значение координаты будет удовлетворять неравенству:

$$[x_k] + 1 \leq x_k \leq \beta_k. \tag{7}$$

Таким образом, образовались две новые ветви, в которых изменилась область допустимых значений для одной неизвестной. Для данной задачи логически возможны следующие случаи:

– Оптимальное решение является целочисленным;

– Оптимальное решение не является целочисленным;

– Данная задача не имеет оптимальных решений.

Разветвление дерева решения в методе ветвей и границ происходит только в том случае, когда оптимальное решение не является целочисленным, в остальных случаях алгоритм прекращает работу.

Недостаток метода ветвей и границ заключается в необходимости решения целочисленной задачи линейного программирования полностью, что отрицательно влияет на время работы алгоритма [11].

Рассмотрим метод ветвей и границ на примере задачи целочисленного линейного программирования. В задаче необходимо найти оптимальное решение.

$$Z = 11x_1 + 5x_2 + 4x_3 \to max,$$
$$\begin{cases} 3x_1 + 2x_2 + 8x_3 \geq 11, \\ 2x_1 + x_3 \leq 5, \\ 3x_1 + 3x_2 + x_3 \leq 13, \\ x_1 \geq 0, x_2 \geq 0, x_3 \geq 0. \end{cases}' \quad . \tag{8}$$
$$x_1, x_2, x_3 - \text{целые числа.}$$

Приведем систему к каноническому виду и решим задачу симплекс-методом, игнорируя целочисленность переменных.

$$Z = 11x_1 + 5x_2 + 4x_3 \to max,$$
$$\begin{cases} 3x_1 + 2x_2 + 8x_3 + x_4 = 11, \\ 2x_1 + x_3 + x_5 = 5, \\ 3x_1 + 3x_2 + x_3 + x_6 = 13, \\ x_i \geq 0, i = 1, \dots, 6. \end{cases}' \quad . \tag{9}$$
$$x_1, x_2, x_3, x_4, x_5, x_6 - \text{целые числа.}$$

Опорное решение $X = (0,0,0,11,5,13)$. Далее необходимо составить симплекс-таблицу (таблица 1).

Таблица 1 – Начальная симплекс-таблица

Базис	Свободный член	x_1	x_2	x_3	x_4	x_5	x_6
x_4	11	3	2	8	1	0	0
x_5	5	2	0	1	0	1	0
x_6	13	3	3	1	0	0	1
Z	0	-11	-5	-4	0	0	0

В пятой строчке таблицы 1 присутствуют отрицательные значения, поэтому необходимо сделать первый шаг симплекс-метода: выбираем столбец с наименьшим значением x_1 и разрешающий элемент в строке x_5.

Составим следующую таблицу для нового шага симплекс-метода (таблица 2).

Таблица 2 – Симплекс-таблица

Базис	Свободный член	x_1	x_2	x_3	x_4	x_5	x_6
x_4	$\frac{7}{2}$	0	2	$\frac{13}{2}$	1	$-\frac{3}{2}$	0
x_1	$\frac{5}{2}$	1	0	$\frac{1}{2}$	0	$\frac{1}{2}$	0
x_6	$\frac{11}{2}$	0	3	$-\frac{1}{2}$	0	$-\frac{3}{2}$	1
Z	$\frac{55}{2}$	0	-5	$\frac{3}{2}$	0	$\frac{11}{2}$	0

В последней оценочной строке до сих пор есть отрицательные числа, следовательно, делаем следующий шаг симплекс-метода. Аналогично первому выбираем значения в столбце с наименьшей оценкой и с разрешающей строкой – это x_2 и x_4 соответственно (таблица 3).

Таблица 3 – Завершающая симплекс-таблица

Базис	Свободный член	x_1	x_2	x_3	x_4	x_5	x_6
x_2	$\frac{7}{4}$	0	1	$\frac{13}{4}$	$\frac{1}{2}$	$-\frac{3}{4}$	0
x_1	$\frac{5}{2}$	1	0	$\frac{1}{2}$	0	$\frac{1}{2}$	0
x_6	$\frac{1}{4}$	0	0	$-\frac{41}{4}$	$-\frac{3}{2}$	$-\frac{3}{4}$	1
Z	$\frac{145}{4}$	0	0	$\frac{71}{4}$	$\frac{5}{2}$	$\frac{7}{4}$	0

В оценочной строке отсутствуют отрицательные числа, следовательно, оптимальное решение найдено: $x_1 = \frac{5}{2}, x_2 = \frac{7}{4}, x_3 = 0, Z_{max} = \frac{145}{4}$.

Оптимальное решение получилось дробное. Далее выбираем переменную с дробным значением, например $x_1 = 5/2 = 2{,}5$, и разбиваем данную задачу на две подзадачи (ветви). В первой задаче полагаем $x_1 \leq 2$, а во второй $x_1 \geq 3$. Решим обе задачи согласно описанному выше алгоритму, добавляя в каждую соответствующее условие для x_1.

Задача 1.

$$Z = 11x_1 + 5x_2 + 4x_3 \to max,$$
$$\begin{cases} 3x_1 + 2x_2 + 8x_3 \geq 11, \\ 2x_1 + x_3 \leq 5, \\ 3x_1 + 3x_2 + x_3 \leq 13, \\ x_1 \leq 2, \\ x_1 \geq 0, x_2 \geq 0, x_3 \geq 0. \end{cases} \quad (10)$$

Данная задача имеет нецелочисленное оптимальное решение: $x_1 = 2, x_2 = \frac{51}{22}, x_3 = \frac{1}{22}, Z_{max} = \frac{743}{22}$.

Задача 2.

$$Z = 11x_1 + 5x_2 + 4x_3 \to max,$$
$$\begin{cases} 3x_1 + 2x_2 + 8x_3 \geq 11, \\ 2x_1 + x_3 \leq 5, \\ 3x_1 + 3x_2 + x_3 \leq 13, \\ x_1 \geq 3, \\ x_1 \geq 0, x_2 \geq 0, x_3 \geq 0. \end{cases} \quad . \tag{11}$$

Задача 2 не имеет решения.

На следующем шаге снова разбиваем задачу 1 на две подзадачи (ветви). Выбираем дробную переменную, например $x_2 = \frac{51}{22} \approx 2{,}3$. В первой задаче полагаем $x_2 \leq 2$, во второй $x_2 \geq 3$. Решаем обе задачи.

Задача 1-1.

$$Z = 11x_1 + 5x_2 + 4x_3 \to max,$$
$$\begin{cases} 3x_1 + 2x_2 + 8x_3 \geq 11, \\ 2x_1 + x_3 \leq 5, \\ 3x_1 + 3x_2 + x_3 \leq 13, \\ x_1 \leq 2, \\ x_2 \leq 2, \\ x_1 \geq 0, x_2 \geq 0, x_3 \geq 0. \end{cases} \quad . \tag{12}$$

Данная задача имеет нецелочисленное оптимальное решение: $x_1 = 2, x_2 = 2, x_3 = \frac{1}{8}, Z_{max} = \frac{65}{2} = 32{,}5$.

Задача 1-2.

$$Z = 11x_1 + 5x_2 + 4x_3 \to max,$$
$$\begin{cases} 3x_1 + 2x_2 + 8x_3 \geq 11, \\ 2x_1 + x_3 \leq 5, \\ 3x_1 + 3x_2 + x_3 \leq 13, \\ x_1 \leq 2, \\ x_2 \geq 3, \\ x_1 \geq 0, x_2 \geq 0, x_3 \geq 0. \end{cases} . \quad (13)$$

Данная задача имеет нецелочисленное оптимальное решение: $x_1 = \frac{9}{7}, x_2 = 3, x_3 = \frac{1}{7}, Z_{max} = \frac{208}{7} \approx 29{,}7.$

Для разбиения на подзадачи необходимо выбрать ту задачу, где оптимальное решение больше, то есть задачу 1-1. Выберем дробную переменную $x_3 = \frac{1}{8} = 0{,}125$, заменяем равенство на неравенства $x_3 \leq 0$ и $x_3 \geq 1$.

Задача 1-1-1.

$$Z = 11x_1 + 5x_2 + 4x_3 \to max,$$
$$\begin{cases} 3x_1 + 2x_2 + 8x_3 \geq 11, \\ 2x_1 + x_3 \leq 5, \\ 3x_1 + 3x_2 + x_3 \leq 13, \\ x_1 \leq 2, \\ x_2 \leq 2, \\ x_3 \leq 0, \\ x_1 \geq 0, x_2 \geq 0, x_3 \geq 0. \end{cases} . \quad (14)$$

Задача 1-1-1 имеет целочисленное оптимальное решение: $x_1 = 2, x_2 = 2, x_3 = 0, Z_{max} = 36.$

Задача 1-1-2.

$$Z = 11x_1 + 5x_2 + 4x_3 \to max,$$
$$\begin{cases} 3x_1 + 2x_2 + 8x_3 \geq 11, \\ 2x_1 + x_3 \leq 5, \\ 3x_1 + 3x_2 + x_3 \leq 13, \\ x_1 \leq 2, \\ x_2 \leq 2, \\ x_3 \geq 1, \\ x_1 \geq 0, x_2 \geq 0, x_3 \geq 0. \end{cases} . \quad (15)$$

Аналогично задаче 1-1-1 задача 1-1-2 имеет целочисленное оптимальное решение: $x_1 = 2, x_2 = 2, x_3 = 1, Z_{max} = 15$.

Для определения целочисленного оптимального решения общей задачи необходимо выбрать наибольшее значение целевой функции – это $Z_{max} = 36$, достигающееся при $x_1 = 2, x_2 = 2, x_3 = 0$. Дерево решений нашей задачи показано на рисунке 1.

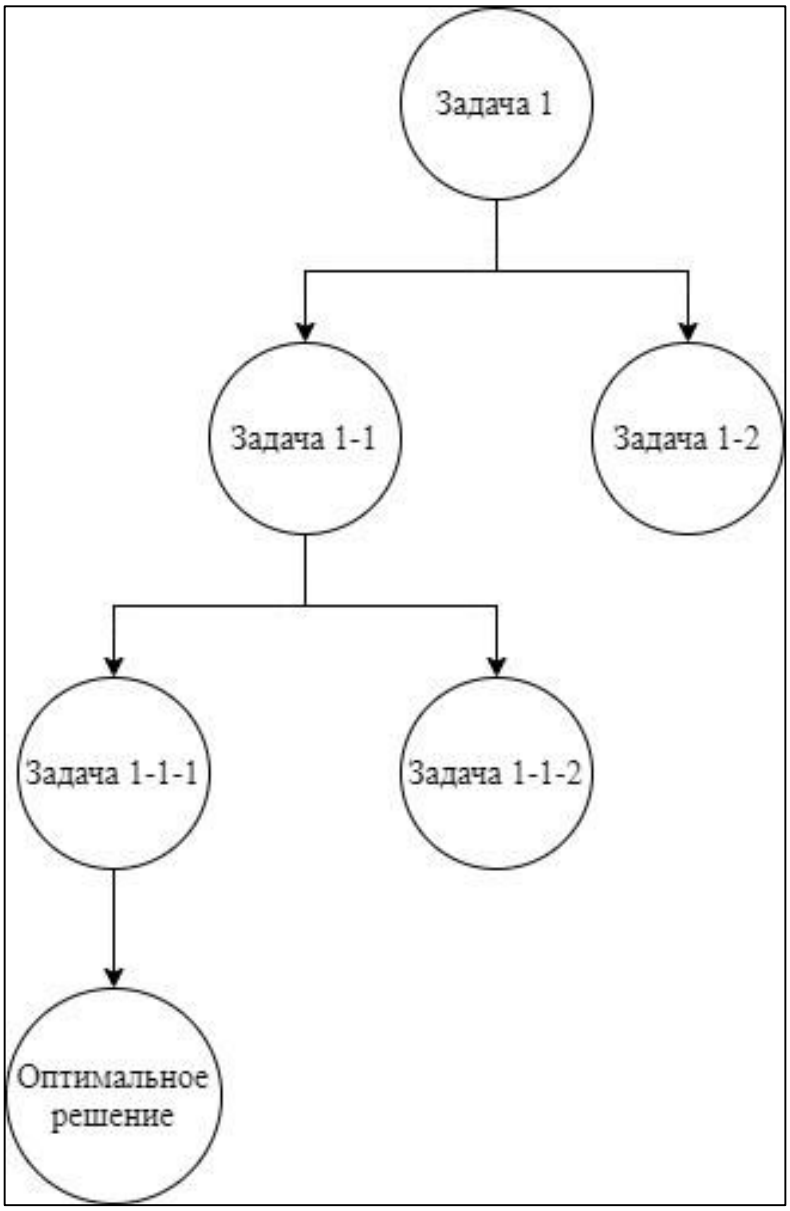

Рисунок 1 – Дерево решений задачи методом ветвей и границ

Итак, был рассмотрен пример решения задачи целочисленного линейного программирования методом ветвей и границ.

1.2.2 Метод Гомори

Метод решения задачи целочисленного и частично целочисленного линейного программирования, предложенный Гомори, основан на симплексном методе [13]. Его суть состоит в решении ослабленной линейной задачи с последующим добавлением линейных ограничений, которые отсекают нецелочисленные решения без отсечения допустимых

целочисленных решений [16]. Данный метод работает следующим образом. Симплексным методом находится оптимальный план задачи без учета условия целочисленности. Если оптимальный план целочисленный, то вычисления заканчиваются, если же оптимальный план содержит хотя бы одну дробную компоненту, то накладывается дополнительное ограничение, учитывающее целочисленность компонент плана, и вычисления симплексным методом продолжаются до получения нового оптимального плана. Если и он является нецелочисленным, то составляются следующие ограничения, учитывающие целочисленность [20].

Процесс присоединения дополнительных ограничений повторяется до тех пор, пока не будет найдено целочисленное оптимальное решение, или пока не будет доказано, что задача не имеет целочисленных решений.

Недостатком метода Гомори является требование целочисленности для всех переменных, как основных, выражающих единицы продукции, так и для дополнительных, выражающих величины неиспользованных ресурсов, которые могут быть дробными [23].

Рассмотрим на примере решение задачи методом Гомори.

$$Z = 3x_1 + x_2 \to max,$$
$$\begin{cases} 2x_1 + 3x_2 \leq 6, \\ 2x_1 - 3x_2 \leq 3, \\ x_{1,2} \in Z^+. \end{cases} \quad (16)$$

Решим данную задачу симплекс-методом без ограничений целочисленности.

$$Z = 3x_1 + x_2 \to max,$$
$$\begin{cases} 2x_1 + 3x_2 \leq 6, \\ 2x_1 - 3x_2 \leq 3, \\ x_{1,2} \geq 0. \end{cases} \quad (17)$$

Приведем задачу к каноническому виду:

$$Z = 3x_1 + x_2 \to max,$$
$$\begin{cases} 2x_1 + 3x_2 + x_3 = 6, \\ 2x_1 - 3x_2 + x_4 = 3, \\ x_{1,2,3,4} \geq 0. \end{cases} . \quad (18)$$

Построим симплекс-таблицу на основе системы 18 (таблица 4).

Таблица 4 – Начальная симплекс-таблица

Базис	Свободный член	x_1	x_2	x_3	x_4
x_3	6	2	3	1	0
x_4	3	2	-3	0	1
Z	0	-3	-1	0	0

В четвертой строке присутствуют отрицательные значения – это означает, что необходимо воспользоваться симплекс-методом. Так как наименьшую оценку имеет столбец x_1, а наименьшее отношение свободных членов строка x_4, то выбираем в качестве разрешавшего элемента их пересечение. Результаты представлены в таблице 5.

Таблица 5 – Симплекс-таблица

Базис	Свободный член	x_1	x_2	x_3	x_4
x_3	3	0	6	1	-1
x_1	$\frac{3}{2}$	-1	$-\frac{3}{2}$	0	$\frac{1}{2}$
Z	$\frac{9}{2}$	0	$-\frac{11}{2}$	0	$\frac{3}{2}$

В последней строке таблицы 5 все еще присутствуют отрицательные значения, поэтому следующим шагом выбираем столбец x_2 и строку x_3 (таблица 6).

Таблица 6 – Заключительная симплекс-таблица

Базис	Свободный член	x_1	x_2	x_3	x_4
x_2	$\frac{1}{2}$	0	1	$\frac{1}{6}$	$-\frac{1}{6}$
x_1	$\frac{9}{4}$	1	0	$\frac{1}{4}$	$\frac{1}{4}$
Z	$\frac{29}{4}$	0	0	$\frac{11}{12}$	$\frac{7}{12}$

В оценочной строке нет отрицательных значений, поэтому оптимальное решение найдено: $x_1 = \frac{9}{4}, x_2 = \frac{1}{2}, Z_{max} = \frac{29}{4}$.

Далее продолжаем решение, используя метод Гомори. Найдем целые части найденного оптимального решения $x_1 = \left[\frac{9}{4}\right] = 2, x_2 = \left[\frac{1}{2}\right] = 0$ и дробные $x_1 = \left\{\frac{9}{4}\right\} = \frac{9}{4} - 2 = \frac{1}{4}, x_2 = \left\{\frac{1}{2}\right\} = \frac{1}{2} - 0 = \frac{1}{2}$.

Выбираем переменную с наибольшей дробной частью – x_2. Введем дополнительное ограничение целочисленности: $0x_2 + \frac{1}{6}x_3 + \frac{5}{6}x_4 \geq \frac{1}{2}$, отсюда $-\frac{1}{6}x_3 - \frac{5}{6}x_4 + x_5 = -\frac{1}{2}$. Получается система:

$$Z = 3x_1 + x_2 \to max,$$
$$\begin{cases} 2x_1 + 3x_2 + x_3 = 6, \\ 2x_1 - 3x_2 + x_4 = 3, \\ -\frac{1}{6}x_3 - \frac{5}{6}x_4 + x_5 = -\frac{1}{2}, \\ x_{1,2,3,4,5} \geq 0. \end{cases} . \quad (19)$$

Составим симплекс-таблицу (таблица 7).

Таблица 7 – Симплекс-таблица, составленная по системе 19

Базис	Свободный член	x_1	x_2	x_3	x_4	x_5
x_2	$\frac{1}{2}$	0	1	$\frac{1}{6}$	$-\frac{1}{6}$	0

Продолжение таблицы 7

Базис	Свободный член	x_1	x_2	x_3	x_4	x_5
x_1	$\frac{9}{4}$	1	0	$\frac{1}{4}$	$\frac{1}{4}$	0
x_5	$-\frac{1}{2}$	0	0	$-\frac{1}{6}$	$-\frac{5}{6}$	1
Z	$\frac{29}{4}$	0	0	$\frac{11}{12}$	$\frac{7}{12}$	0

Перейдем к следующему шагу (таблица 8).

Таблица 8 – Симплекс-таблица

Базис	Свободный член	x_1	x_2	x_3	x_4	x_5
x_2	$\frac{6}{10}$	0	1	$\frac{1}{5}$	0	$-\frac{1}{5}$
x_1	$\frac{21}{10}$	1	0	$\frac{1}{5}$	0	$\frac{3}{10}$
x_4	$\frac{3}{5}$	0	0	$\frac{1}{5}$	1	$-\frac{6}{5}$
Z	$\frac{69}{10}$	0	0	$\frac{4}{5}$	0	$\frac{7}{10}$

По таблице 8 получилось нецелочисленное решение. Выбираем переменную x с наибольшей дробной частью, то есть x_2. В x_2 дробная часть равна $\frac{3}{5}$.

Введем еще одно дополнительное ограничение целочисленности: $\frac{1}{5}x_3 + \frac{4}{5}x_4 \geq \frac{3}{5}$, отсюда $-\frac{1}{5}x_3 - \frac{4}{5}x_4 + x_6 = -\frac{3}{5}$. Добавим полученное ограничение в систему.

$$Z = 3x_1 + x_2 \to max,$$
$$\begin{cases} 2x_1 + 3x_2 + x_3 = 6, \\ 2x_1 - 3x_2 + x_4 = 3, \\ -\frac{1}{6}x_3 - \frac{5}{6}x_4 + x_5 = -\frac{1}{2}, \\ -\frac{1}{5}x_3 - \frac{4}{5}x_4 + x_6 = -\frac{3}{5}, \\ x_{1,2,3,4,5,6} \geq 0. \end{cases} . \quad (20)$$

И составим симплекс-таблицу (таблица 9).

Таблица 9 – Симплекс-таблица, составленная по системе 20

Базис	Свободный член	x_1	x_2	x_3	x_4	x_5	x_6
x_2	$\frac{6}{10}$	0	1	$\frac{1}{5}$	0	$-\frac{1}{5}$	0
x_1	$\frac{21}{10}$	1	0	$\frac{1}{5}$	0	$\frac{3}{10}$	0
x_4	$\frac{3}{5}$	0	0	$\frac{1}{5}$	1	$-\frac{6}{5}$	0
x_6	$-\frac{7}{10}$	0	0	$-\frac{1}{5}$	0	$-\frac{4}{5}$	1
Z	$\frac{69}{10}$	0	0	$\frac{4}{5}$	0	$\frac{7}{10}$	0

Перейдем к следующему шагу (таблица 10).

Таблица 10 – Симплекс-таблица

Базис	Свободный член	x_1	x_2	x_3	x_4	x_5	x_6
x_2	$\frac{3}{4}$	0	1	$\frac{1}{4}$	0	0	$-\frac{1}{4}$
x_1	$\frac{15}{8}$	1	0	$\frac{1}{8}$	0	0	$\frac{3}{8}$
x_4	$\frac{3}{2}$	0	0	$\frac{1}{52}$	1	0	$-\frac{3}{2}$
x_5	$\frac{3}{4}$	0	0	$\frac{1}{4}$	0	1	$-\frac{5}{4}$

Продолжение таблицы 10

Базис	Свободный член	x_1	x_2	x_3	x_4	x_5	x_6
Z	$\frac{51}{8}$	0	0	$\frac{5}{8}$	0	0	0

Получили нецелочисленное решение. Выбираем переменную x_1, потому что у нее наибольшая дробная часть, которая равна $\frac{7}{8}$.

Введем дополнительное ограничение в систему: $\frac{1}{8}x_3 + \frac{3}{8}x_6 \geq \frac{7}{8}$, отсюда $-\frac{1}{8}x_3 - \frac{3}{8}x_6 + x_7 = -\frac{7}{8}$. Добавим полученное ограничение в систему.

$$Z = 3x_1 + x_2 \to max,$$
$$\begin{cases} 2x_1 + 3x_2 + x_3 = 6, \\ 2x_1 - 3x_2 + x_4 = 3, \\ -\frac{1}{6}x_3 - \frac{5}{6}x_4 + x_5 = -\frac{1}{2}, \\ -\frac{1}{5}x_3 - \frac{4}{5}x_4 + x_6 = -\frac{3}{5}, \\ -\frac{1}{8}x_3 - \frac{3}{8}x_6 + x_7 = -\frac{7}{8}, \\ x_{1,2,3,4,5,6,7} \geq 0. \end{cases} . \quad (21)$$

С помощью системы 21 составим симплекс-таблицу (таблицу 11).

Таблица 11 – Симплекс-таблица, составленная по системе 21

Базис	Свободный член	x_1	x_2	x_3	x_4	x_5	x_6	x_7
x_2	$\frac{3}{4}$	0	1	$\frac{1}{4}$	0	0	$-\frac{1}{4}$	0
x_1	$\frac{15}{8}$	1	0	$\frac{1}{8}$	0	0	$\frac{3}{8}$	0
x_4	$\frac{3}{2}$	0	0	$\frac{1}{52}$	1	0	$-\frac{3}{2}$	0
x_5	$\frac{3}{4}$	0	0	$\frac{1}{4}$	0	1	$-\frac{5}{4}$	0

Продолжение таблицы 11

Базис	Свободный член	x_1	x_2	x_3	x_4	x_5	x_6	x_7
x_7	$-\frac{7}{8}$	0	0	$-\frac{1}{8}$	0	0	$-\frac{3}{8}$	1
Z	$\frac{51}{8}$	0	0	$\frac{5}{8}$	0	0	0	0

Следующий шаг в таблице 12.

Таблица 12 – Симплекс-таблица

Базис	Свободный член	x_1	x_2	x_3	x_4	x_5	x_6	x_7
x_2	$\frac{4}{3}$	0	1	$\frac{1}{3}$	0	0	0	$-\frac{2}{3}$
x_1	1	1	0	0	0	0	0	1
x_4	5	0	0	1	1	0	0	-4
x_5	$\frac{11}{3}$	0	0	$\frac{2}{3}$	0	1	0	$-\frac{10}{3}$
x_6	$\frac{7}{3}$	0	0	$\frac{1}{3}$	0	0	1	$-\frac{8}{3}$
Z	$\frac{13}{3}$	0	0	$\frac{1}{3}$	0	0	0	$\frac{7}{3}$

Получили нецелочисленное решение, поэтому выбираем снова переменную с наибольшей дробной частью (x_5 – дробная часть равна $\frac{2}{3}$).

Введем дополнительное ограничение в систему: $\frac{2}{3}x_3 + \frac{2}{3}x_7 \geq \frac{2}{3}$, отсюда $-\frac{2}{3}x_3 - \frac{2}{3}x_7 + x_8 = -\frac{2}{3}$. Аналогично предыдущим шагам добавим полученное ограничение в систему.

$$Z = 3x_1 + x_2 \to max,$$

$$\begin{cases} 2x_1 + 3x_2 + x_3 = 6, \\ 2x_1 - 3x_2 + x_4 = 3, \\ -\frac{1}{6}x_3 - \frac{5}{6}x_4 + x_5 = -\frac{1}{2}, \\ -\frac{1}{5}x_3 - \frac{4}{5}x_4 + x_6 = -\frac{3}{5}, \\ -\frac{1}{8}x_3 - \frac{3}{8}x_6 + x_7 = -\frac{7}{8}, \\ -\frac{2}{3}x_3 - \frac{2}{3}x_7 + x_8 = -\frac{2}{3}, \\ x_{1,2,3,4,5,6,7,8} \geq 0. \end{cases} \quad . \qquad (22)$$

Таблица 13 – Симплекс-таблица, составленная по системе 22

Базис	Свободный член	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
x_2	$\frac{4}{3}$	0	1	$\frac{1}{3}$	0	0	0	$-\frac{2}{3}$	0
x_1	1	1	0	0	0	0	0	1	0
x_4	5	0	0	1	1	0	0	-4	0
x_5	$\frac{11}{3}$	0	0	$\frac{2}{3}$	0	1	0	$-\frac{10}{3}$	0
x_6	$\frac{7}{3}$	0	0	$\frac{1}{3}$	0	0	1	$-\frac{8}{3}$	0
x_8	$-\frac{2}{3}$	0	0	$-\frac{2}{3}$	0	0	0	$-\frac{2}{3}$	1
Z	$\frac{51}{8}$	0	0	$\frac{5}{8}$	0	0	0	0	0

Перейдем к следующему шагу (таблица 14).

Таблица 14 – Симплекс-таблица

Базис	Свободный член	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
x_2	1	0	1	0	0	0	0	-1	$\frac{1}{2}$
x_1	1	1	0	0	0	0	0	1	0
x_4	4	0	0	0	1	0	0	-5	$\frac{3}{2}$

Продолжение таблицы 14

Базис	Свободный член	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
x_5	3	0	0	0	0	1	0	-4	1
x_6	2	0	0	0	0	0	1	-3	$\frac{1}{2}$
x_3	1	0	0	1	0	0	0	1	$-\frac{3}{2}$
Z	4	0	0	$\frac{1}{3}$	0	0	0	2	$\frac{1}{2}$

Получаем оптимальное целочисленное решение $x_1 = 1, x_2 = 1, Z_{max} = 4$. Таким образом, мы рассмотрели на примере решение задачи методом Гомори.

В данном разделе были проанализированные два основных точных метода для решения целочисленных или частично целостных задач линейного программирования.

2 Реализация алгоритмов для решения целочисленных и частично-целочисленных задач линейного программирования

2.1 Структура реализованной программы

Алгоритм метода ветвей и границ и метод Гомори были реализованы на языке программирования Python в интегрированной среде разработки $PyCharm$, которая была создана международной компанией $JetBrains$.

Для реализации алгоритмов были использованы следующие библиотеки:

– $numpy$ 1.16.4;

– $tkinter$;

– $math$;

– $time$;

– $threading$;

– re;

– $random$.

Обзор важных составляющих реализованной программы.

а) парсер входных данных;

1) парсер функции с граничными условиями;
2) парсер функции без граничных условий.

б) генерация уравнений целочисленной и частично-целочисленной задачи линейного программирования;

1) генерация левой части уравнения;
2) генерация правой части уравнения.

в) реализация однопоточного алгоритма метода ветвей и границ решения целочисленных задач линейного программирования;

г) реализация параллельного алгоритма метода ветвей и границ решения целочисленных задач линейного программирования;

д) реализация симплекс-метода решения целочисленных задач линейного программирования;

е) построение дерева решений;

ж) реализация пользовательского графического интерфейса.

2.2 Реализация метода ветвей и границ и метода Гомори

Для решения методом ветвей и границ целочисленной или частично-целочисленной задачи линейного программирования необходимо реализовать парсер входных данных. Они поступают через $grid$ в пользовательском графическом интерфейсе. Составим таблицу состояний (конечный автомат) для парсера (рисунок 2).

```
# Таблица состояний (КА)
__STATES = [
    [1, 2, 'US', 'US', 5, 'UF'],          # 0 — Вход в конечный автомат
    ['US', 2, 'US', 'US', 5, 'UF'],       # 1 — Знак перед числом (+/-)
    ['US', 2, 3, 4, 5, 'EP'],             # 2 — Ввод числа
    ['US', 2, 'US', 'US', 'US', 'UF'],    # 3 — Ввод точки в числе перед иксом
    ['US', 'US', 'US', 'US', 5, 'UF'],    # 4 — Ввод умножения
    ['US', 6, 'US', 'US', 'US', 'UF'],    # 5 — Ввод икса
    [0, 6, 'US', 'US', 'US', 'EP']        # 6 — Ввод номера икса
]
```

Рисунок 2 – Таблица состояний

Также сделаем расшифровку ошибок (рисунок 3).

```
# Расшифровки к ошибкам:
# US — Unexpected symbol (Неожиданный символ)
# UF — Unfinished formula (Незаконченная формула)
# UC - Unknown character (Неизвестный символ)
# BCE — No boundary conditions (Отсутсвуют граничные условия)
__EXCEPTIONS = {
    "US": "Неожиданный символ '{}'!",
    "UF": "Формула не закночена!",
    "UC": "Неизвестный символ '{}'!",
    "BCE": "Отсутсвуют граничные условия или неверный разделитель!"
}
```

Рисунок 3 – Расшифровка ошибок

В файл $Parser.py$ входят два класса $Parser$ и $ParserException$. Последний класс нужен для отлавливания и вывода ошибок.

Класс $Parser$ реализован в программе и необходим как раз для того, чтобы обрабатывать входные данные, которые поступают через пользовательский графический интерфейс. В данном классе реализованы следующие функции:

- $parse\_function\_with\_borders$ – функция, необходимая для того, чтобы можно было распарсить целевую функцию с граничными условиями;
- $parse\_alone\_function$ – функция, необходимая для того, чтобы можно было распарсить целевую функцию без граничных условий;
- $find\_max\_constant$ – функция, необходимая для того, чтобы можно было распарсить целевую функцию и все ее ограничения;

Далее идут приватные функции:

- $\_\_parse\_side\_of\_function$ – парсинг одной части целевой функции;
- $\_\_is\_needed\_to\_save\_cof$ – проверка на необходимость обнулить буфер;
- $\_\_is\_need\_update\_cof\_buffer$ – проверка на необходимость записать значение из буфера в буфер коэффициентов;
- $\_\_is\_need\_to\_save\_char$ – проверка на необходимость записать текущий символ в буфер (это происходит только при запоминании чисел);
- $\_\_is\_contains\_borders$ – функция, проверяющая, содержит ли строка граничный знак;
- $\_\_split\_formula$ – делит строку по граничному знаку $« > », « < », « >= », « <= », « = »$;
- $\_\_resolve\_to\_float$ – перевод строки в число.

Диаграмма класса Parser представлена на рисунке 4.

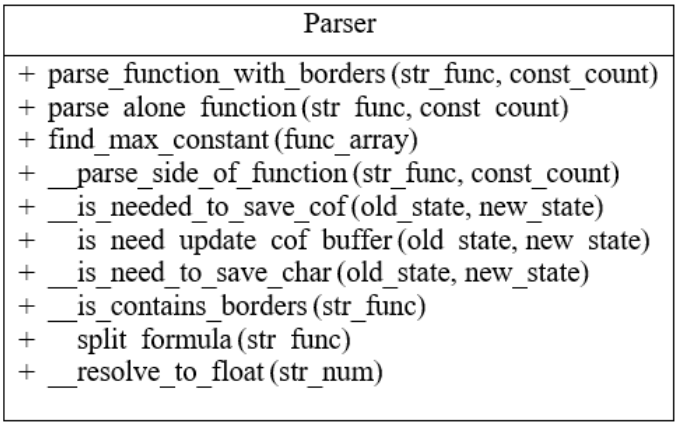

Рисунок 4 – UML-диаграмма класса $Parser$

Рассмотрим, какие параметры принимает на вход каждая из реализованных функций в классе $Parser$.

Начнем с $parse\_function\_with\_borders$. Данная функция принимает на вход два параметра – функцию в виде строки (str_func) и количество используемых констант ($const\_count$). Также на выход функция отправляет два массива, первый состоит из коэффициентов векторов для левой и правой части дерева решений, а второй из свободных членов для левой и правой части дерева решений. В самой функции $parse\_function\_with\_borders$ сначала проверяется, если ли граничные условия (рисунок 5).

```
if not Parser.__is_contains_borders(str_func):
    raise ParserException(Parser.__EXCEPTIONS["BCE"])
```

Рисунок 5 – Проверка на граничные условия

Далее задаются левые и правые части целевой функции (рисунок 6).

```
left_side, right_side, border = Parser.__split_formula(str_func)

left_cof, left_free_cof = Parser.__parse_side_of_function(left_side, const_count)
right_cof, right_free_cof = Parser.__parse_side_of_function(right_side, const_count)
```

Рисунок 6 – Задание левой и правой частей целевой функции

Затем в исходном коде идет функция $parse\_alone\_function$. Она принимает на вход два параметра: str_func – функцию в виде строки и $const\_count$ – количество используемых констант. На выход функция выдает вектор коэффициентов и свободный член.

В функции $find\_max\_constant$ на вход принимается лишь одно значение – массив с целевой функцией и всеми заданными ограничениями ($func\_array$), а выдает функция номер наибольшего x.

Рассмотрим функцию $\_\_parse\_side\_of\_function$. На вход подаются следующие два параметра: str_func и $const\_count$, где первый параметр – это часть функции в виде строки, а второй – количество используемых констант. Данная функция возвращает те же данные, что и $parse\_alone\_function$, то есть вектор коэффициентов и свободный член.

Функции $\_\_is\_needed\_to\_save\_cof$, $\_\_is\_need\_update\_cof\_buffer$ и $\_\_is\_need\_to\_save\_char$ принимают на вход old_state и new_state, старое и новое состояние соответственно. Выходные данные у этих функций одинаковые – это булевый параметр $True/\ False$, где $True$ – обновить буфер (записать символ в буфер), а $False$ – не обновлять буфер (не записывать число в буфер).

У функций $\_\_is\_contains\_borders$, $\_\_split\_formula$ и $\_\_resolve\_to\_float$ входной параметр один и тот же, а выходные параметры разные. Входной параметр – str_func – функция в виде строки. Выходной параметр для первой функции – это булевый параметр $True/\ False$, где $True$ – строка содержит граничный знак, а $False$ – строка не содержит граничный знак. В функции $\_\_split\_formula$ на выходе передаются левая и правая части функции, разделяющий знак. А функция $\_\_resolve\_to\_float$ переводит подаваемое на вход число из строки в формат $float$.

Реализация класса $SimplexMethodSolver$ для нахождения решения симплекс методом осуществляется посредством следующих функций:

– $find\_solution$ – поиск решения симплекс-методом;

– $\_\_check\_to\_normal$ – проверка на возможность решения.

В функцию $find\_solution$ поступают следующие параметры: $function$ – функция, $bounds\_array$ – ограничения функции и $optimal\_model$ – параметр в виде булевой переменной, где $True$ – это функция на максимум, а $False$ – функция на минимум.

Функция $\_\_check\_to\_normal$ зависит от следующих параметров: $coefficients$ – массив коэффициентов, $arguments$ – массив аргументов, $borders$ – массив условий и $variables$ – массив значений x.

На этом реализация симплекс-метода закончена, полный код программы приведен в приложении А.

Представим диаграмму класса $SimplexMethodSolver$ (рисунок 7).

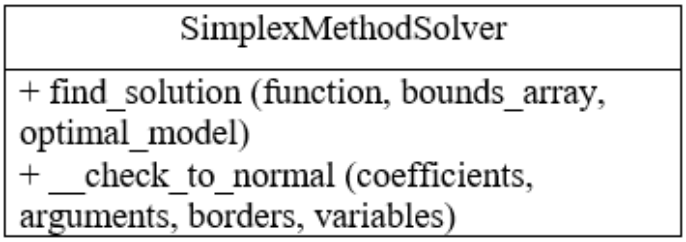

Рисунок 7 – UML-диаграмма класса $SimplexMethodSolver$

Для генерации целочисленной или частично-целочисленной задачи линейного программирования используется класс $GeneratorUtils$. В данный класс входят следующие три функции:

– $generate\_liner\_condition$ – генерация ЦЛП задачи;

– $\_\_generate\_left\_part$ – генерация левой части уравнения;

– $\_\_generate\_right\_part$ – генерация правой части уравнения.

Рассмотрим каждую функцию отдельно. В первой и во второй функциях на вход подаются следующие параметры:

– num_of_args – количество аргументов;

– left_border – левая граница генерируемых чисел;

– right_border – правая граница генерируемых чисел.

Первая функция возвращает сгенерированные ограничения и целевую функцию.

Функция $\_\_generate\_left\_part$ возвращает левую часть сгенерированного уравнения.

В последней функции $\_\_generate\_right\_part$ подаются два параметра: $left\_border$ – левая граница генерируемых чисел и $right\_border$ – правая граница генерируемых чисел. На выходе из функции приходит правая часть целевой функции.

На рисунке 8 представлена диаграмма класса $GeneratorUtils$.

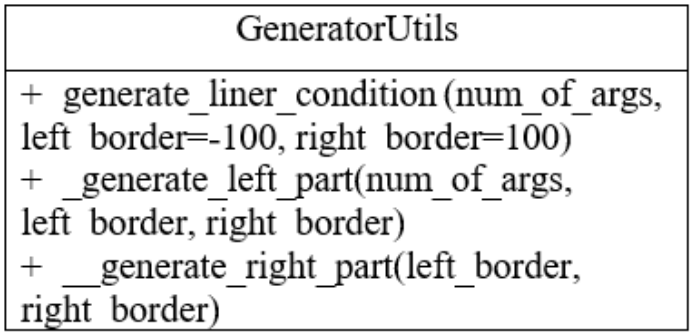

Рисунок 8 – UML-диаграмма класса $GeneratorUtils$

Далее создается класс $BranchAndBorderMethod$ – это класс для решения целочисленной задачи линейного программирования методом ветвей и границ. В данный класс входят следующие функции:

- $find\_solution\_with\_all\_params$ – поиск решения и получение дополнительной информации о решении;
- $find\_solution\_with\_time$ – поиск решения и получение времени работы метода ветвей и границ;
- $find\_solution$ – поиск решения;

Далее идут приватные функции:

- $\_\_get\_solution$ – поиск решения;
- $\_\_find\_solution\_by\_multiple\_threads$ – поиск решения в многопоточном режиме;

– $\_\_find\_solution\_by\_single\_thread$ – поиск решения в однопоточном режиме;
– $\_\_recount\_node$ – пересчет значений в узле дерева решений;
– $get\_not\_integer\_var$ – поиск дробного числа из списка всех полученных значений x;
– $\_\_is\_contains\_nulls$ – функция для обнаружения нулевых значений в полученном решении;
– $create\_new\_bounds$ – создание дополнительных ограничений (выполняется только со второй итерации решения целочисленной и частично-целочисленной задачи линейного программирования методом ветвей и границ);
– $\_\_get\_not\_equal\_bound$ – получение уникальной границы;
– $\_\_get\_max\_depth$ – получение высоты дерева;
– $\_\_find\_min$ – поиск наименьшего решения;
– $\_\_find\_max$ – поиск наибольшего решения;
– $\_\_get\_all\_nodes$ – получение всех узлов найденного дерева решений;
– $\_\_get\_all\_results$ – получение всех узлов дерева, содержащих решение.

В функцию $\_\_get\_solution$ поступают такие параметры как:
– $expect\_nulls$ – параметр, который говорит о том, ожидает ли программа нулевые значения;
– max_depth – максимальная высота дерева;
– $function$ – целевая функция;
– $bounds\_array$ – массив ограничений;
– $optimal\_model$ – булевый параметр, который принимает значение $True$ или $False$, где $True$ – это поиск решения на максимум, а $False$ – поиск решения на минимум;
– $is\_multiple\_threads$ – булевый параметр, который принимает значение $True$ или $False$, где $True$ – поиск решения в

многопоточном режиме, а $False$ – поиск решения в однопоточном режиме.

Функции $\_\_find\_solution\_by\_multiple\_threads$ и $\_\_find\_solution\_by\_single\_thread$ принимают параметры, схожие с параметрами функции, описанной выше, поэтому описание данных параметров не требуется. Но, в отличие от $\_\_get\_solution$, функции на поиск решения в многопоточном и однопоточном режиме возвращают дополнительные параметры: наибольшее или наименьшее значение целевой функции для решения целочисленной или частично-целочисленной задачи линейного программирования, коэффициенты xn, время работы функции и глубину дерева.

Далее рассмотрим входные параметры $\_\_recount\_node$: max_depth – максимальная глубина дерева решений, $node$ – узел дерева. Функция возвращает значение в виде дерева решений.

Функции $get\_not\_integer\_var$ и $\_\_is\_contains\_nulls$ принимают на вход одинаковый параметр $variables$ — массив значений xn. Но функция $get\_not\_integer\_var$ возвращает номер x, который имеет дробный коэффициент, а $\_\_is\_contains\_nulls$ определяет, содержит ли массив нулевые значения: $True$ – содержит, $False$ – не содержит.

В функцию для создания дополнительных ограничений ($create\_new\_bounds$) на вход подаются параметры num_of_x – номер x и $value$ – значение коэффициента. Выходным параметром функции являются границы для левого и правого узла дерева.

Функция $\_\_get\_not\_equal\_bound$ принимает следующие параметры: $bound$ – новая граница, $bounds\_array$ – существующие границы. Если новая граница уникальна, то возвращается она сама, иначе параметр $None$.

В функциях $\_\_get\_max\_depth$, $\_\_find\_min$ и $\_\_find\_max$ подается на вход параметр $root$ – корневой узел, но в последних двух функциях на вход подается еще один параметр – $expect\_nulls$ – ожидание нулевых значений.

Входные параметры функции __get_all_nodes следующие: $node$ – узел, $result$ – массив всех узлов.

Последняя функция в классе $BranchAndBorderMethod$ – это $\_\_get\_all\_results$. Она принимает на вход значения узлов ($node$), булевую переменную для ожидания нулевого решения в узлах дерева решений ($expect\_nulls$) и массив всех узлов решения ($result$).

Представим диаграмму класса $BranchAndBorderMethod$ (рисунок 9).

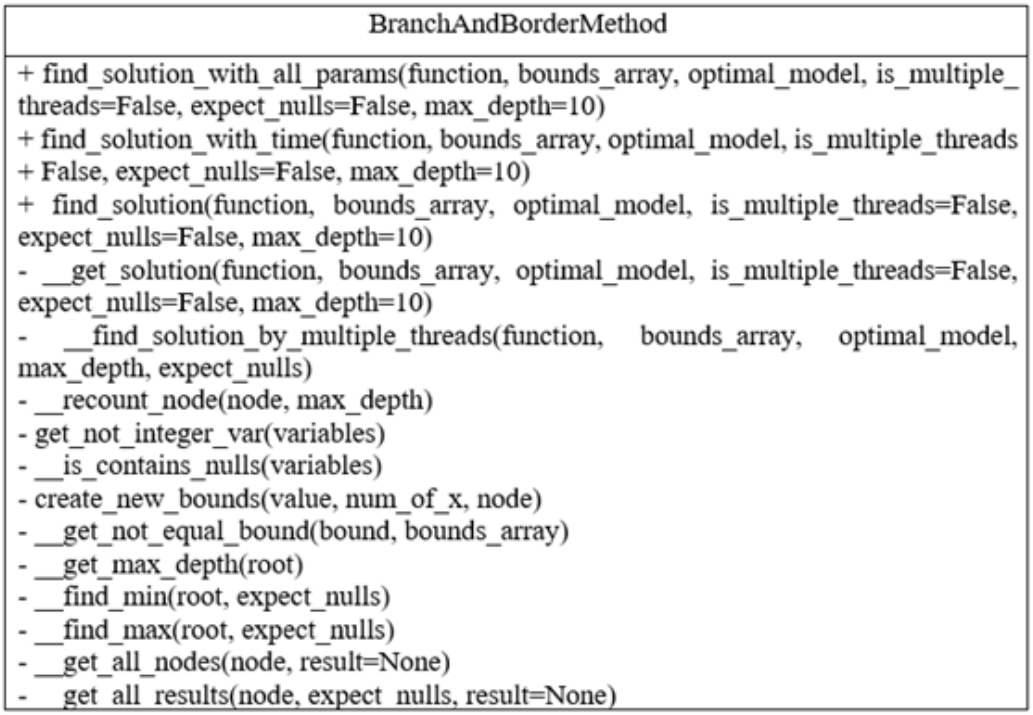

Рисунок 9 – UML-диаграмма класса $BranchAndBorderMethod$

Для метода ветвей и границ необходимы классы $Node$ – класс, описывающий узел в дереве решений и $Tree$ – класс, описывающий дерево решений.

Класс $Node$ состоит из следующих функций:

– $\_\_init\_\_$ – конструктор класса;

– $set\_results$ – вывод результатов;

– set_left – функция присвоения значений в левых узлах;

– set_right – функция присвоения значений в правых узлах;

– get_left – получение значений левых узлов;

– get_right – получение значений правых узлов;

– get_bounds – функция для получения ограничений на данном узле;

– $get\_function$ – получение целевой функции;

– get_model – получение параметра MAX/MIN;

– $\_\_str\_\_$ – вывод текущего состояния объекта.

Класс $Tree$ состоит только из конструктора класса.

Диаграммы классов $Node$ и $Tree$ представлены на рисунках 10 и 11 соответственно.

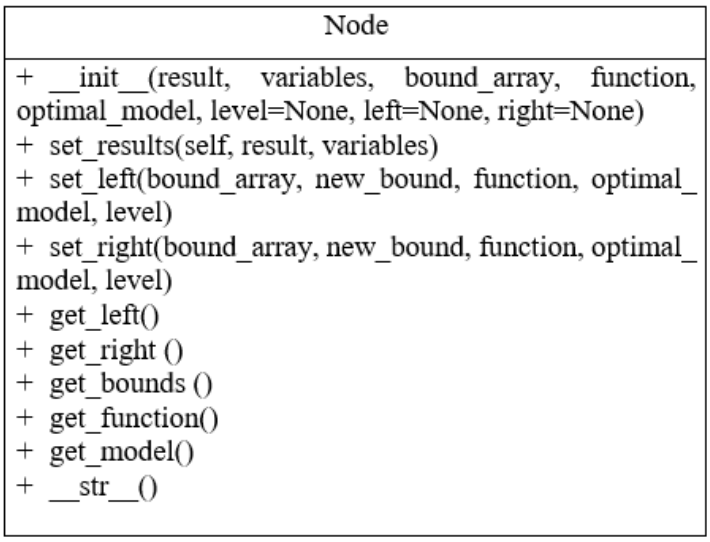

Рисунок 10 – UML-диаграмма класса $Node$

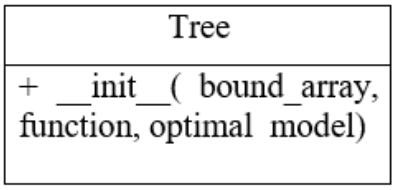

Рисунок 11 – UML-диаграмма класса $Tree$

На этом реализация алгоритмов завершена, перейдем к пользовательскому графическому интерфейсу.

2.3 Пользовательский графический интерфейс приложения

Для удобного использования реализованной программы был разработан пользовательский графический интерфейс с помощью встроенной библиотеки $tkinter$. Ниже приведен пример работы пользовательского графического интерфейса для алгоритма ветвей и границ и для алгоритма Гомори рисунок 12 и 13 соответственно.

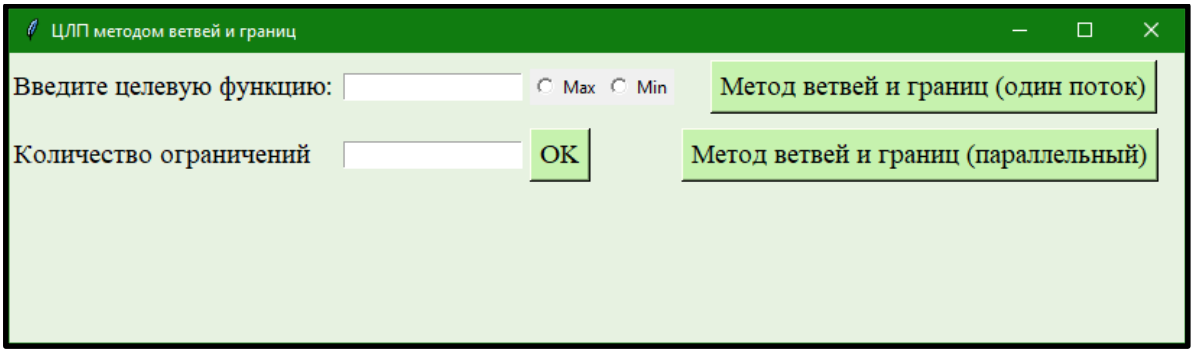

Рисунок 12 – Пример работы пользовательского графического интерфейса для алгоритма ветвей и границ

Рисунок 13 – Пример работы пользовательского графического интерфейса для алгоритма Гомори

В исходных файлах программы есть файл $Form.py$. Он создан для реализации пользовательского графического интерфейса. В этом файле присутствуют следующие глобальные переменные:

– $limitations\_text$ – массив ограничений в виде строки;
– $limitations\_entries$ – массив ограничений для вывода;
– $borders\_labels$ – массив порядковых номеров ограничений.

В $Form.py$ есть следующие функции:

– $number\_of\_limitations$ – функция для создания $labels$ в пользовательском графическом интерфейсе;
– BnP – функция, решающая целочисленную или частично-целочисленную задачу линейного программирования методом ветвей и границ в одном потоке, вызывая при этом класс $BranchAndBorderMethod$;
– BnP_parall – функция, аналогичная функции BnP, но для решения целочисленных или частично-целочисленных задач линейного программирования в нескольких потоках.
– $CutPln$ – функция, решающая целочисленную или частично-целочисленную задачу линейного программирования методом Гомори в одном потоке, вызывая при этом класс $CuttingPlaneMethods$;
– $CutPln\_parall$ – функция, аналогичная функции $CutPln$, но для решения целочисленных или частично-целочисленных задач линейного программирования в нескольких потоках.

Окно интерфейса для решения ЦЛП методом ветвей и границ задаётся на рисунке 14.

```
bg = '#e7f2e1'
root = Tk()
root.configure(background=bg)
root.title("ЦЛП методом ветвей и границ")
root.geometry("1000x200")
```

Рисунок 14 – Функции для создания окна интерфейса

Далее при помощи встроенной библиотеки $tkinter$ и параметра $label$ мы создаем в форме строку «Введите целевую функцию». Также через параметр $grid$ выводим ее на пользовательский графический интерфейс (рисунок 15).

```
target_function_str = StringVar()
target_function = Label(text="Введите целевую функцию:",
                        fg="#000000", bg=bg, font='Times 14')
target_function.grid(row=0, column=0, sticky="w")
```

Рисунок 15 – Создание $label$ и вывод с помощью $grid$

Ввод целевой функции происходит через пользовательский графический интерфейс. Выбор направления оптимизации производится посредством элемента $radio\ button$ (рисунок 16).

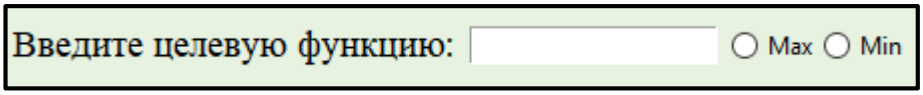

Рисунок 16 – Ввод целевой функции

$Radio\ buttons$ были реализованы через встроенную функцию $tkinter.Radiobutton$ (рисунок 17).

```
check = IntVar()
max_radiobutton = ttk.Radiobutton(text="Max", style='Wild.TRadiobutton',
                                  value=1, variable=check)
max_radiobutton.grid(row=0, column=2, sticky="w")
min_radiobutton = ttk.Radiobutton(text="Min", style='Wild.TRadiobutton',
                                  value=2, variable=check)
min_radiobutton.grid(row=0, column=3, sticky="w")
```

Рисунок 17 – Ввод целевой функции

Далее задается количество ограничений в целочисленной или частично-целочисленной задаче линейного программирования. После нажатия на кнопку «Ок» необходимо в каждый $grid$ ввести сами ограничения (рисунок 18).

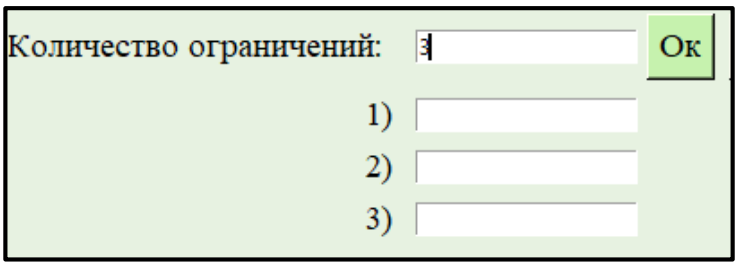

Рисунок 18 – Ввод ограничений

Ограничения могут содержать знаки « > », « < », « >= », « <= », « = » в зависимости от условий целочисленной или частично-целочисленной задачи линейного программирования.

На рисунке 19 представлена реализация полей ввода ограничений и кнопки подтверждения «Ок».

```
count_limitations_str = StringVar()
count_limitations = Label(text="Количество ограничений:",
                          fg="#000000", bg=bg, font='Times 14')
count_limitations.grid(row=1, column=0, sticky="w")
count_limitations_entry = Entry(textvariable=count_limitations_str)
count_limitations_entry.grid(row=1, column=1, padx=5, pady=5)

entering_limitations = Button(text="Ок", fg="#000000", bg="#c6f2ae",
                              font='Times 14', command=number_of_limitations)
entering_limitations.grid(row=1, column=2, sticky="w")
```

Рисунок 19 – Реализация полей ввода ограничений и кнопки подтверждения

Следующим шагом были выведены две кнопки выбора режима решения задачи (в одном потоке или параллельно) целочисленного или частично-целочисленного линейного программирования. Кнопки в форме изображены на рисунке 20 и 21.

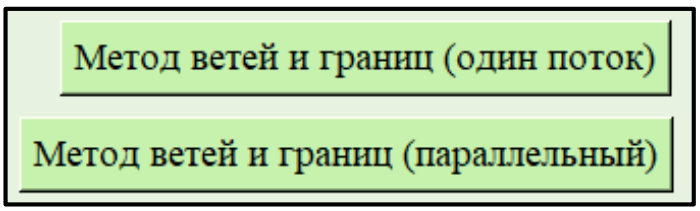

Рисунок 20 – Кнопки в форме

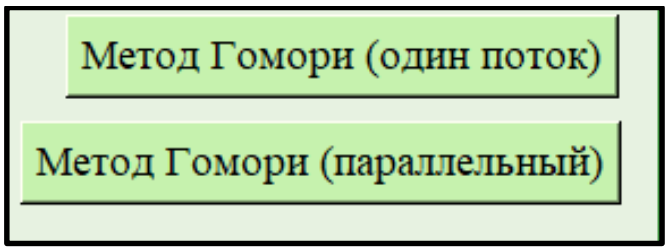

Рисунок 21 – Кнопки в форме

Реализация данных кнопок представлена на рисунке 22.

```
bnp_ordinary = Button(text="Метод ветей и границ (один поток)",
                      fg="#000000", bg="#c6f2ae", font='Times 14',
                      command=BnP)
bnp_ordinary.grid(row=0, column=4, padx=25, pady=5, sticky="w")
bnp_ordinary.entry = Label(padx=0, pady=0, font='Times 14', bg=bg)
bnp_ordinary.entry.grid(row=0, column=5, padx=5, pady=5)

bnp_parallel = Button(text="Метод ветей и границ (параллельный)",
                      fg="#000000", bg="#c6f2ae", font='Times 14',
                      command=BnP_parall)
bnp_parallel.grid(row=1, column=4, padx=5, pady=5, sticky="w")
bnp_parallel.entry = Label(padx=0, pady=0, font='Times 14', bg=bg)
bnp_parallel.entry.grid(row=1, column=5, padx=5, pady=5)
```

Рисунок 22 – Реализация кнопок

Программный код пользовательского интерфейса для метода ветвей и границ находится в Приложении Б.

Осталось вывести необходимую для ответа информацию на пользовательский графический интерфейс, а именно: время работы программы и решение целочисленной или частично-целочисленной задачи линейного программирования (рисунок 23).

```
#  Время работы алгоритма
time = Label(padx=0, pady=0, font='Times 14', bg="#e7f2e1")
time.grid(row=1, column=0, sticky=W, columnspan=3)
#  Сумма построенного маршрута
summs = Label(padx=0, pady=0, font='Times 14', bg=bg)
summs.grid(row=2, column=0, sticky=W, columnspan=3)
```

Рисунок 23 – Вывод ответа в форму

Проверим работу реализованного приложения на примере метода ветвей и границ и метода Гомори (рисунок 24 и 25).

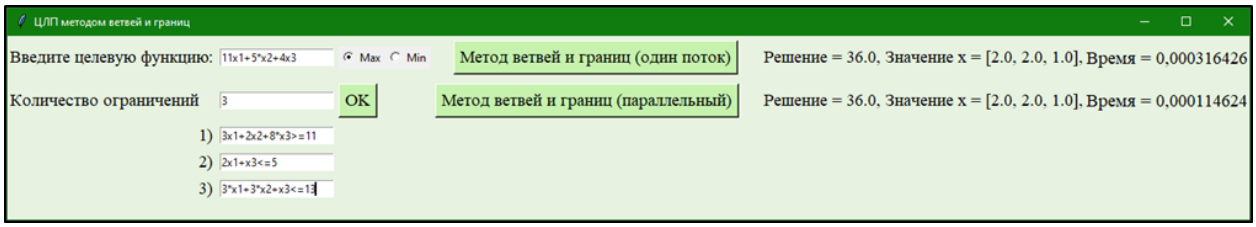

Рисунок 24 – Пример работы приложения для алгоритма ветвей и границ

Рисунок 25 – Пример работы приложения для алгоритма Гомори

В этом разделе был реализован пользовательский графический интерфейс и программная реализация метода Гомори и метода ветвей и границ, а также приведены примеры работы этих методов на тестовых данных.

В следующем разделе будет проведен сравнительный анализ компьютерной реализации методов, разработанных в текущем разделе, с замерами времени.

3 Сравнительный анализ реализаций

3.1 Технические данные для сравнительного анализа

Для проведения сравнительного анализа был выбран персональный компьютер со следующими техническими характеристиками (таблица 15).

Таблица 15 – Технические характеристики персонального компьютера

Процессор	IntelCore i5-6600 CPU 3.30 GHz
Память	16,0 Gb
Операционная система	Microsoft Windows 10 Pro

Результатом работы разработанной программы будет таблица и график со временем работы алгоритма в секундах. Также возможно провести сравнительный анализ полученного времени для обоих алгоритмах будут использоваться одинаковые ограничения.

3.2 Метод ветвей и границ

Написанная ранее программа позволяет задать количество ограничений, которые можно сгенерировать случайным образом. В программе фиксировалось время работы алгоритма ветвей и границ. Исследования проводились от 1 до 1000 ограничений, результаты представлены ниже в таблице 16.

Таблица 16 – Результат фиксирования времени однопоточного алгоритма

Количество ограничений	Время работы однопоточного алгоритма ветвей и границ (секунды)
5	0,00051
10	0,0017
15	0,0048

Продолжение таблицы 16

Количество ограничений	Время работы однопоточного алгоритма ветвей и границ (секунды)
20	0,00899
25	0,01637
30	0,02671
35	0,04085
40	0,06163
45	0,08351
50	0,11281
100	0,84384
200	2,36346
300	3,15128
400	6,95794
500	10,63642
1000	20,4982

Представим данные из таблицы 16 в виде графика (рисунок 26).

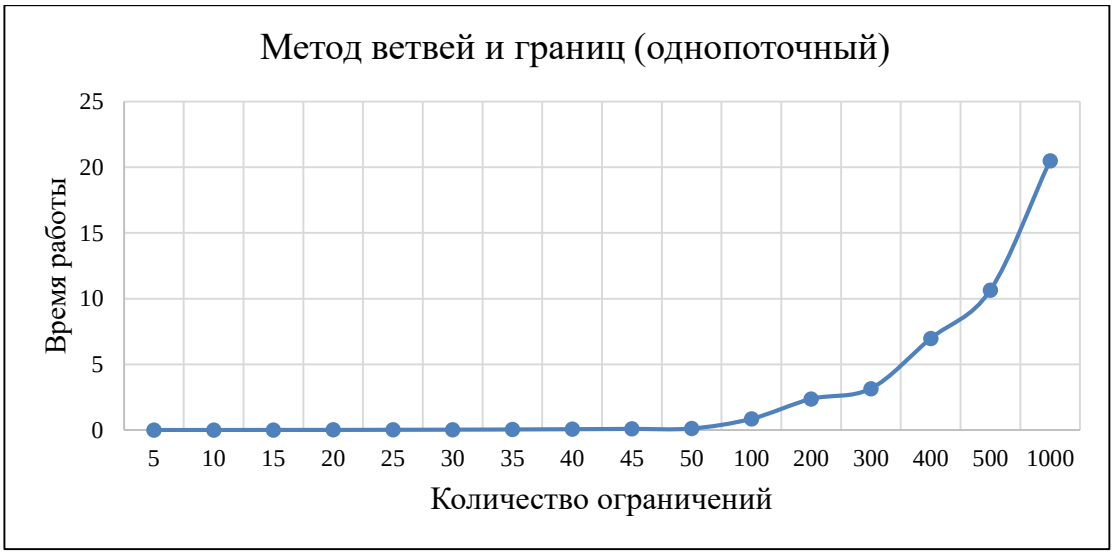

Рисунок 26 – График зависимости времени от количества ограничений

Благодаря графику (рисунок 26) мы видим, что на малых количествах ограничений время работы алгоритма ветвей и границ затрачивает достаточно малое количество времени.

Аналогично таблице 16 сделаем замеры времени в параллельном алгоритме ветвей и границ. Количество ограничений будут использованы абсолютно такие же, как и при замерах времени в однопоточном алгоритме (таблица 17).

Таблица 17 – Результат фиксирования времени параллельного алгоритма

Количество ограничений	Время работы параллельного алгоритма ветвей и границ (секунды)
5	0,00013
10	0,00024
15	0,00061
20	0,000146
25	0,000246
30	0,000438
35	0,000746
40	0,001134
45	0,001573
50	0,004358
100	0,034689
200	0,165796
300	1,597616
400	2,971351
500	7,469713
1000	14,56489

Представим данные из таблицы 17 в виде графика (рисунок 27).

Рисунок 27 – График зависимости времени от количества ограничений

Исходя из графика на рисунке 27, можно сделать вывод, что при увеличении количества ограничений время работы параллельного алгоритма значительно растет вверх.

3.3 Сравнение графиков однопоточной и параллельной реализации алгоритма ветвей и границ

Для наглядного примера объединим из пункта 3.2 полученные результаты в виде графиков в один. Аналогичную операцию сделаем и с таблицей с замерами времени работы алгоритма ветвей и границ. Общие результаты работы двух реализаций представлены в таблице 18.

Таблица 18 – Результат фиксирования времени параллельного алгоритма

Количество ограничений	Время работы однопоточного алгоритма ветвей и границ (секунды)	Время работы параллельного алгоритма ветвей и границ (секунды)
5	0,00051	0,00013
10	0,0017	0,00024

Продолжение таблицы 18

Количество ограничений	Время работы однопоточного алгоритма ветвей и границ (секунды)	Время работы параллельного алгоритма ветвей и границ (секунды)
15	0,0048	0,00061
20	0,00899	0,000146
25	0,01637	0,000246
30	0,02671	0,000438
35	0,04085	0,000746
40	0,06163	0,001134
45	0,08351	0,001573
50	0,11281	0,004358
100	0,84384	0,034689
200	2,36346	0,165796
300	3,15128	1,597616
400	6,95794	2,971351
500	10,63642	7,469713
1000	20,4982	14,56489

Далее представим объединенные результаты в виде графика (рисунок 28).

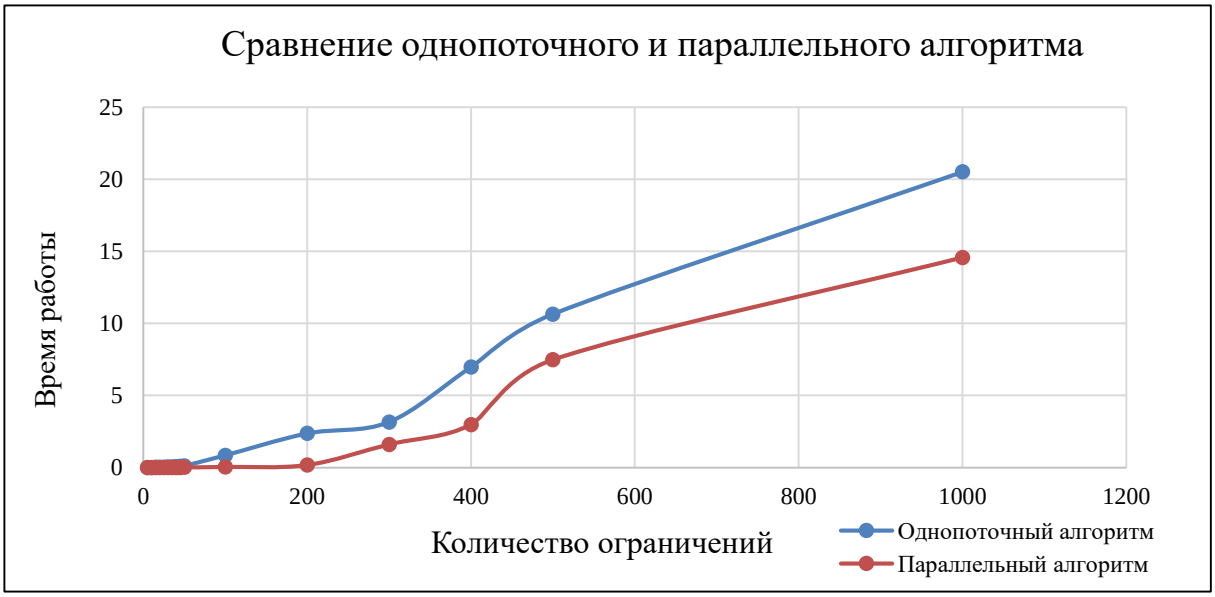

Рисунок 28 – Сравнение однопоточного и параллельного алгоритма

Из рисунка 28 мы наглядно видим, что однопоточный и параллельный алгоритм по затратам времени на работу алгоритмы ветвей и границ одинаково возрастают на больших количествах ограничений.

Из этого можно сделать вывод, что при решении поставленных задач целочисленного или частично целочисленного линейного программирования в практических задачах стоит отдавать предпочтение параллельному алгоритму, нежели однопоточному.

3.4 Метод Гомори

Проведем сравнение времени реализованного однопоточного и параллельного алгоритма Гомори для целочисленного или частично целочисленного линейного программирования, аналогично методу ветвей и границ в пункте 3.1.

Для сравнения времени возьмем количество ограничений от 1 до 1000, результаты приставлены в таблице 19 ниже.

Таблица 19 – Результат фиксирования времени однопоточного алгоритма

Количество ограничений	Время работы однопоточного алгоритма Гомори (секунды)
5	0,000059
10	0,000227
15	0,000315
20	0,000680
25	0,000773
30	0,001077
35	0,001437
40	0,001748
45	0,002623
50	0,002625
100	0,010576
200	0,040370
300	0,094620

Продолжение таблицы 19

Количество ограничений	Время работы однопоточного алгоритма Гомори (секунды)
400	0,147421
500	0,232296
1000	1,109630

Представим данные из таблицы 19 в виде графика (рисунок 29).

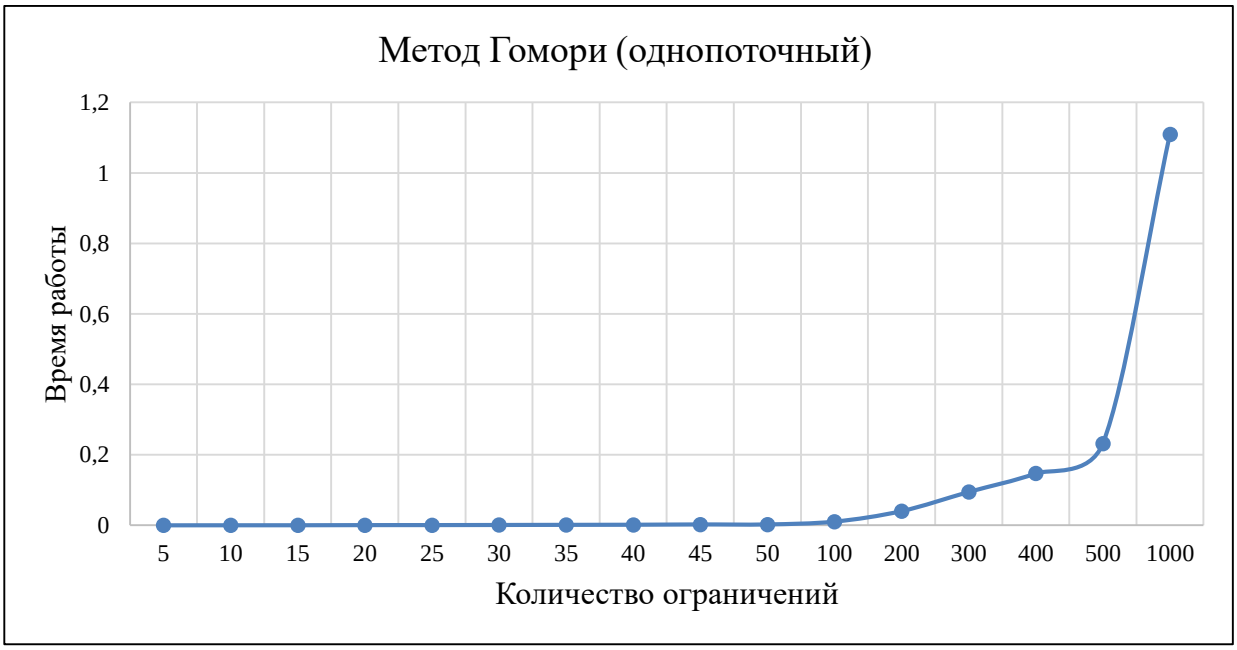

Рисунок 29 – График зависимости времени от количества ограничений

Благодаря графику (рисунок 29) мы видим, что на малых количествах ограничений время работы алгоритма ветвей и границ затрачивает очень мало времени.

Аналогично таблице 19 сделаем замеры для параллельного алгоритма Гомори. Будет использовано такое же количество ограничений, как и при замерах в однопоточном алгоритме (таблица 20).

Таблица 20 – Результат фиксирования времени параллельного алгоритма

Количество ограничений	Время работы параллельного алгоритма Гомори (секунды)
5	0,000040
10	0,000132
15	0,000214
20	0,000361
25	0,000426
30	0,000604
35	0,000732
40	0,000869
45	0,001046
50	0,001354
100	0,001735
200	0,002640
300	0,007521
400	0,013698
500	0,046821
1000	0,189461

Представим данные из таблицы 20 в виде графика (рисунок 30).

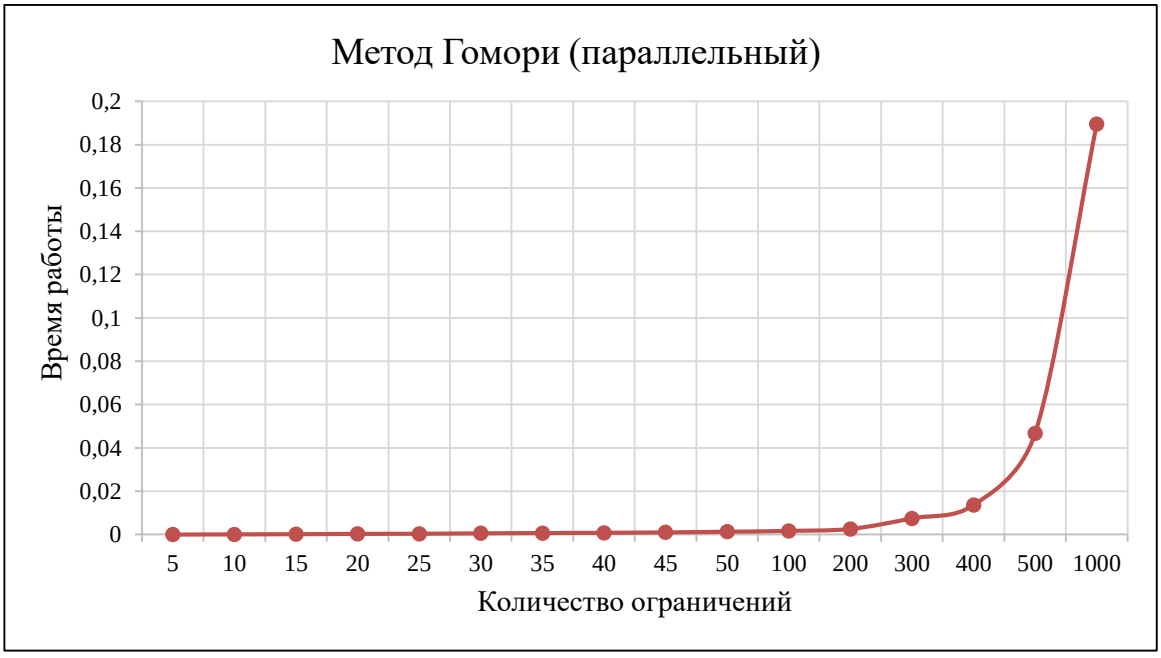

Рисунок 30 – График зависимости времени от количества ограничений

Исходя из графика на рисунке 30, можно сделать вывод, что при количестве ограничений больше 100 время работы параллельного алгоритма значительно повышается.

3.5 Сравнение графиков однопоточной и параллельной реализации алгоритма Гомори

Для наглядного примера объединим полученные результаты из пункта 3.4 в одну таблицу. Результаты объединения результатов представлены в таблице 21.

Таблица 21 – Результат фиксирования времени параллельного алгоритма

Количество ограничений	Время работы однопоточного алгоритма Гомори (секунды)	Время работы параллельного алгоритма Гомори (секунды)
5	0,000059	0,000040
10	0,000227	0,000132
15	0,000315	0,000214
20	0,000680	0,000361
25	0,000773	0,000426
30	0,001077	0,000604
35	0,001437	0,000732
40	0,001748	0,000869
45	0,002623	0,001046
50	0,002625	0,001354
100	0,010576	0,001735
200	0,040370	0,002640
300	0,094620	0,007521
400	0,147421	0,013698
500	0,232296	0,046821
1000	1,109630	0,189461

Далее представим объединенные результаты в виде графика, как это делали в пункте 3.2 (рисунок 31).

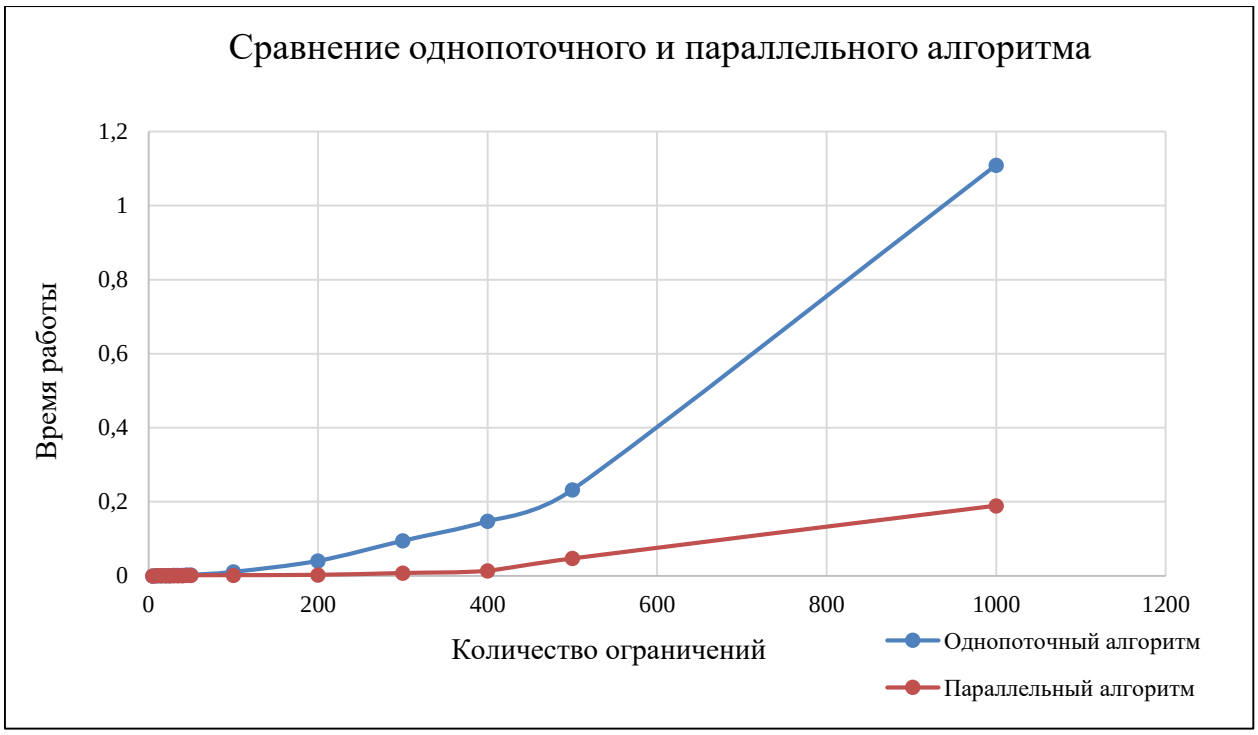

Рисунок 31 – Сравнение однопоточного и параллельного алгоритма

Из рисунка 31 мы видим, что параллельный алгоритм намного превосходит однопоточный. Хотя на малых количествах ограничений расхождения во времени работы алгоритма незначительные, но для больших задач и проектов требуется большое количество ограничений.

Можно сделать вывод, что если компаниям или производствам потребуется использовать целочисленные или частично целочисленные задачи линейного программирования методом Гомори в сфере производственного планирования или распределения, то им следует применять параллельный алгоритм, так как он затрачивает значительно меньше времени работы, в отличие от однопоточной реализации алгоритма Гомори.

3.6 Сравнение параллельной реализации алгоритма ветвей и границ и алгоритма Гомори

Из пунктов 3.3 и 3.5 делаем вывод, что нет смысла сравнивать однопоточные реализации алгоритмов, так как они значительно отстают по времени работы алгоритма от параллельной реализации.

Поэтому в данном подпункте будет произведено сравнение двух параллельных реализаций алгоритмов. В таблице 22 объединим время работы алгоритма ветвей и границ и алгоритма Гомори.

Таблица 22 – Результат фиксирования времени параллельного алгоритма

Количество ограничений	Время работы параллельного алгоритма ветвей и границ (секунды)	Время работы параллельного алгоритма Гомори (секунды)
5	0,00013	0,000040
10	0,00024	0,000132
15	0,00061	0,000214
20	0,000146	0,000361
25	0,000246	0,000426
30	0,000438	0,000604
35	0,000746	0,000732
40	0,001134	0,000869
45	0,001573	0,001046
50	0,004358	0,001354
100	0,034689	0,001735
200	0,165796	0,002640
300	1,597616	0,007521
400	2,971351	0,013698
500	7,469713	0,046821
1000	14,56489	0,189461

На основе таблицы 22 представим график, который изображен на рисунке 32.

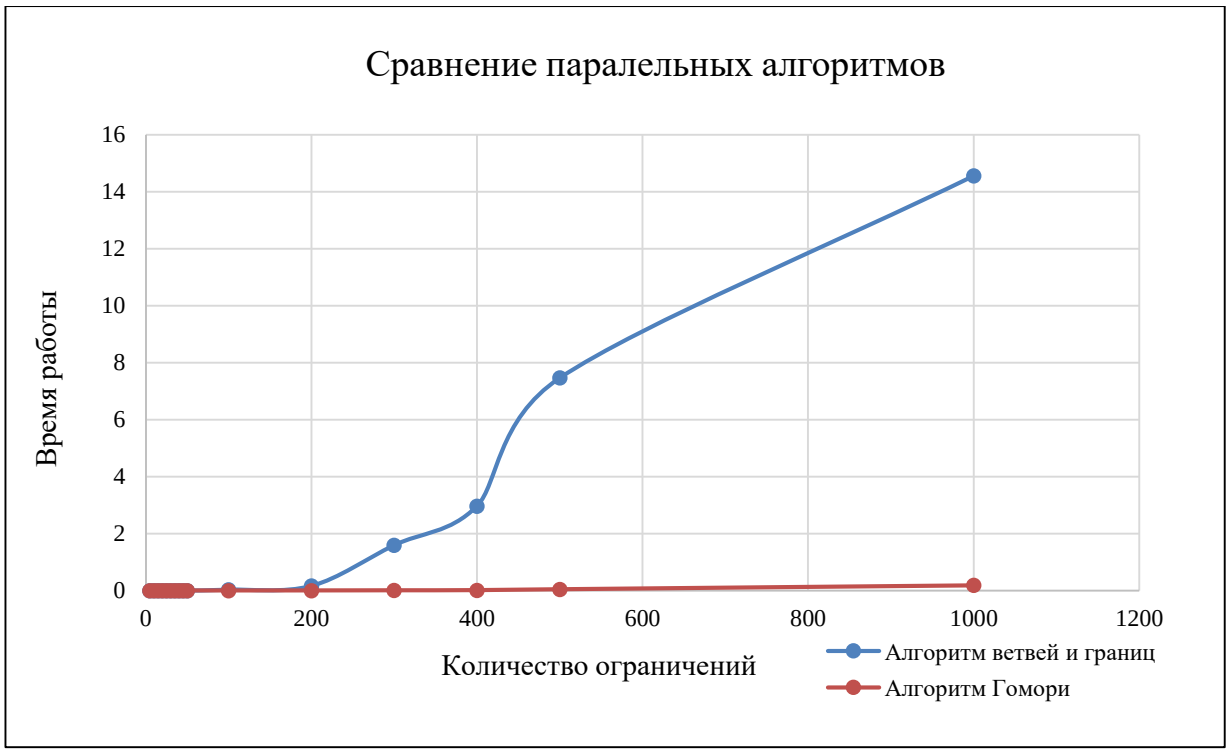

Рисунок 32 – Сравнение параллельных алгоритмов

Из рисунка 32 мы очевидно видим разницу между двумя параллельными алгоритмами. Алгоритм Гомори значительней превосходит по времени работы алгоритма ветвей и границ.

Из этого можно сделать вывод, что различным компаниям или производствам следует обратить внимание именно на алгоритм Гомори. Данный метод намного быстрее справиться с поставленной задачей, например, распределения грузов, чем метод ветвей и границ.

4 Моделирование данных

4.1 Описание математической модели

Рассмотренные ранее условия были сгенерированы случайным образом. Из-за этого результаты выполнения алгоритмов на сухих данных могут отличаться от использования алгоритма на реальных задачах.

Для сравнительного анализа на данных приближенных к реальным задачам можно воспользоваться алгоритмом повышения заработной платы в компании своим сотрудникам с учетом нескольких параметров, предложенным Cokyasar T. [14].

Подготовим необходимые данные для использования математической модели. Этот процесс может быть осуществлен при сотрудничестве отдела кадров компании. Таблица 23 демонстрирует пример собранных данных.

Таблица 23 – Пример данных

N	C	R	U	L	E	P	K
1	5900	7600	8000	6000	3	0	6
2	9000	11000	12000	8500	2	1	3
3	10500	13600	14000	10000	2	1	4
4	7500	8700	10000	7500	5	0	2
5	9100	11800	12000	8500	4	0	4
6	6100	7600	8000	6000	5	1	2
7	6700	6500	7900	5000	4	1	5
8	5800	5900	7800	6200	3	1	1
9	7900	6100	300	8000	2	0	4
10	9600	12000	14000	10000	5	1	2

Описание заголовков таблицы 23 приведено ниже:

– N – Количество сотрудников;

– C – Текущая заработная плата сотрудников;

– R – Преобладающая заработная плата сотрудников;

- U – Верхняя граница заработной платы сотрудников;
- L – Нижняя граница заработной платы сотрудников;
- E – Результаты эволюции производительности, собранные отделом кадров;
- P – Возможность повышения сотрудникам заработной платы;
- K – Количество месяцев, прошедших с момента последнего повышения заработной платы.

Перед формулированием математической модели необходимо для начала принять во внимание следующие предположения:

- верхняя граница зарплаты для каждого сотрудника должна быть выше, чем текущая заработная плата соответствующего сотрудника;
- нижняя граница заработной платы для каждого сотрудника должна быть ниже текущей зарплаты соответствующего сотрудника;
- сотрудников с неудовлетворительными результатами работы не следует включать во входные данные.

Основная цель данной модели – это переназначить руководящих сотрудников к оптимальным категориям повышения заработной платы, чтобы минимизировать общую разницу между их новой заработной платой и текущей, соответствующей их статусу. Таким образом, руководящие сотрудники уже распределяются по категориям заработной платы на основе результатов оценки их работы соответствующим отделом, обычно это отдел кадров. Получается, что использование слова «переназначить» означает распределение сотрудников по категориям повышения заработной платы на основе результатов оценки их работы с некоторыми дополнительными ограничениями. Например, компания может захотеть отнести одного или нескольких своих сотрудников к более низкой заработной плате, чем текущая, потому что общая сумма зарплаты, выплачиваемой персоналу, может быть выше, чем общая сумма преобладающих окладов. Таким образом, компания

может убедиться, что не платит своим сотрудникам больше, чем их конкуренты.

Переменные:

- X_m – Процент увеличения, соответствующий категории m;
- Z_{im} – Равно 1, если для работника i задан режим заработной платы m, иначе равно 0 и W_{it} – умножение X_m и Z_{im};
- λ_i – Дополнительная переменная линеаризации;
- δ_i – Дополнительная переменная линеаризации.

Индексы:

- $i = 1,2,\dots,N$;
- $m = 1,2,\dots,M$.

Параметры:

- N – Количество квалифицированного руководящего состава для повышения заработной платы;
- M – Количество категорий повышения заработной платы;
- E_i – Текущий результат оценки работы сотрудника i;
- C_i – Текущая зарплата сотрудника i;
- R_i – Преобладающая заработная плата для должности сотрудника i;
- U_i – Верхняя граница заработная плата сотрудника i;
- L_i – Нижняя граница заработная плата сотрудника i;
- P_i – Потенциал сотрудника i для продвижения по службе. Предполагается, что потенциал равен 0 ил 1 для каждого сотрудника;
- a_1 – Нижняя граница процентного увеличения заработной платы $(0 \le a_1 \le 1)$;
- a_2 – Верхняя граница процентного увеличения заработной платы $(0 \le a_2 \le 1)$;
- a_3 – Минимальная разница между процентами увеличения зарплаты $(0 \le a_3 \le a_2)$;

– d – Максимальное отклонение в возрастных категориях, допустимое между производительностью работы и заработной платы;
– q – Максимальная позиция в диапазоне заработной платы, разрешения для низко потенциального персонала ($0 \leq q \leq 1$);

Математическая формулировка:

$$\sum_i \left| C_i \left(\sum_m Z_{im} X_m + 1 \right) - R_i \right| \rightarrow min. \quad (23)$$

При условии:

$$C_i \left(\sum_m Z_{im} X_m + 1 \right) \leq P_i U_i + (1 - P_i)[L_i + q(U_i - L_i] \text{ , для всех } i, \quad (24)$$

$$C_i \left(\sum_m Z_{im} X_m + 1 \right) \geq L_i, \text{для всех } i, \quad (25)$$

$$X_1 \geq a_1, X_m \leq a_2, X_M - X_{M-1} \geq a_3, \text{для всех } m \geq 2, \quad (26)$$

$$\sum_m m Z_{im} \leq e_i, \text{для всех } i, \quad (27)$$

$$\sum_m m Z_{im} \leq e_i - d, \text{для всех } i, \quad (28)$$

$$\sum_m Z_{im} = 1, \text{для всех } i, \quad (29)$$

$$Z_{im} = 0{,}1, \text{для всех } i \text{ и } m. \quad (30)$$

Целевая функция (формула 23) направлена на минимизацию общего отклонения (включая как положительную, так и отрицательную разницу) между всеми новыми зарплатами руководящего персонала и всеми текущими заработными платами. Поскольку целевая функция включает в себя

умножение двух переменных, данная модель является нелинейной математической моделью.

Ограничения в формуле 24 и 25 устанавливают верхние границы для всех сотрудников в зависимости от их потенциала. Таким образом, эти два ограничения нацелены на выполнение условий внутренней согласованности. Компании, использующие эту математическую модель, могут в основном устанавливать верхнюю и нижнюю границу в зависимости от своего бюджета. Потенциал ограничения (формула 24) влияет на верхнюю границу заработной платы каждого сотрудника. Например, если потенциальное значение сотрудника равно 1, то он получает в качестве зарплаты верхнюю границу [31]. В противоположной ситуации, если у сотрудника нулевой (или низкий) потенциал, то верхняя граница новой зарплаты сотрудника умножается назначение от 0 до 1, чтобы уменьшить верхнюю границу и позволить сотруднику получить повышение зарплаты в пределах полученной границы. Следовательно, вознаграждение сотрудников также обеспечивается за счет влияния на их новую заработную плату, основанную на их высоком потенциале, что может быть справедливым применением в конкурентной рабочей среде [14].

Набор ограничений (формула 26) создает интервал для повышения категорий заработной платы. Первое ограничение в наборе гарантирует, что самая низкая граница повышения зарплаты должна составлять, по крайней мере, определённый диапазон, определенный компанией (пользователем) модели. Второе ограничение указывает верхнюю границу повышения заработной платы. Следовательно, такая категория зарплаты не может превышать самый высокий процент увеличения заработной платы, который компания желает установить [25]. Последний набор ограничений определяет минимальный процент между соседними категориями повышения заработной платы. Следовательно, разница между категориями повышения зарплаты будет мотивировать сотрудников приносить больше пользы компании.

Ограничение (формула 27) гарантирует, что сотрудники, подлежащие рассмотрению для повышения заработной платы, не должны быть отнесены к категории производительности выше, чем их заранее определенные результаты производительности. Это ограничение определяет диапазон новых назначений зарплаты для сотрудников. Таким образом, эта формула 28 ограничивает руководство по отнесению сотрудника к чрезвычайно низкой заработной плате. Другими словами, предполагая $d = 1$, компания подтверждает, что все сотрудники, рассматриваемые в модели, будут отнесены либо к их заранее определенной категории, либо на одну категорию ниже, чем текущая. Увеличение d может увеличить диапазон, таким образом, что можно будет создать больше возможностей для набора решений [15].

Ограничения из формулы 29 и 30 гарантируют, что каждому сотруднику, включенному в руководство, будет назначена только одна категория для повышения заработной платы.

В качестве возврата в данной модели получаются значения X и Z. Значение X – это категории увеличения заработной платы, которые определяют процент увеличения зарплаты каждого сотрудника. Параметры a_1 и a_2 определяют границы между самым низким и самым высоким процентным значением X. Таким образом, модель определят X процентов на основе внутренней согласованности компаний (пользователя). С другой стороны, переменная Z демонстрируют соответствие сотрудников своим категориям оптимального повышения заработной платы. Другими словами, индексы переменных Z иллюстрируют оптимальную категорию повышения зарплаты для каждого сотрудника. Например, Z_{13} означает, что первый сотрудник отнесен к категории для повышения заработной платы 3. Если X_3 имеет оптимальное значение 11%, то можно сделать вывод, что первый сотрудник в рассмотрении получит повышение в зарплате на 11% в соответствии с представленной моделью. На этом этапе было проиллюстрирована модель в

нелинейной форме, также объясняются все ограничения и их роль в модели назначения категории для повышения заработной платы [17].

Процедура линеаризации.

Существует несколько методов линеаризации нелинейной целевой функции и ограничений. Один из этих методов – линеаризация минимизации суммы абсолютных значений [12]. Согласно их методу, новую форма целевой функции можно проиллюстрировать следующим образом:

$$\sum_i \lambda_i + \delta_i \text{ , для всех } i. \tag{31}$$

Таким образом, линеаризованная целевая функция становится ограничением, при условии:

$$C_i \left(\sum_m Z_{im} X_m + 1 \right) - \lambda_i + \delta_i = R_i, \text{для всех } i, \tag{32}$$

где $\lambda_i \geq 0$ и $\delta_i \geq 0$, для всех i

Нелинейное выражение можно линеаризовать, умножив на новую переменную, которое покрывает умножение обеих переменных. В результате этого процесса ниже показано следующее преобразование:

$$W_{im} = Z_{im} X_m, \text{для всех } i \text{ и } m. \tag{33}$$

После принятия во внимание вышеупомянутого преобразования добавляются следующие ограничения:

$$W_{im} \leq Z_{im}, \text{для всех } i \text{ и } m, \tag{34}$$

$$W_{im} \geq Z_{im} + X_m - 1, \text{для всех } i \text{ и } m, \tag{35}$$

$$W_{im} \leq X_m, \text{для всех } i \text{ и } m. \tag{36}$$

После применения всего этого процесса линеаризации модель превратилась в частично целочисленную задачу линейного программирования [18]. Поэтому окончательная линеаризованная форма частично целочисленной задачи линейного программирования проиллюстрирована ниже:

$$\sum_m \lambda_i + \delta_i \rightarrow min, \text{для всех } i. \tag{37}$$

При условии:

$$C_i\left(\sum_m Z_{im}X_m + 1\right) - \lambda_i + \delta_i = R_i, \text{для всех } i, \tag{38}$$

$$C_i\left(\sum_m Z_{im}X_m + 1\right) \leq P_iU_i + (1 - P_i)[L_i + q(U_i - L_i)], \text{для всех } i, \tag{39}$$

$$C_i\left(\sum_m Z_{im}X_m + 1\right) \geq L_i, \text{для всех } i, \tag{40}$$

$$X_1 \geq a_1, X_m \leq a_2, X_M - X_{M-1} \geq a_3, \text{для всех } m \geq 2, \tag{41}$$

$$\sum_m mZ_{im} \leq e_i, \text{для всех } i, \tag{42}$$

$$\sum_m mZ_{im} \leq e_i - d, \text{для всех } i, \tag{43}$$

$$\sum_m Z_{im} = 1, \text{для всех } i, \tag{44}$$

$$W_{im} \leq Z_{im}, \text{для всех } i \text{ и } m, \tag{45}$$

$$W_{im} \geq Z_{im} + X_m - 1, \text{для всех } i \text{ и } m, \tag{46}$$

$$W_{im} \leq X_m, \text{для всех } i \text{ и } m, \tag{47}$$

$$Z_{im} = 0{,}1, \text{для всех } i \text{ и } m, \tag{48}$$

$$X_m \geq 0, W_{im} \geq 0, Z_{im} \geq 0, \lambda_i \geq 0, \text{и } \delta_i \geq 0, \text{для всех } i \text{ и } m. \tag{49}$$

Поскольку целевая функция модели является функцией минимизации, дальнейшие операции не требуются. С другой стороны, некоторые ограничения, такие как в формулах 40, 41 и 46, должны быть умножены на -1, чтобы заменить знаки больше или равно на знаки меньше или равно. Более того, максимальное упрощение неравенств также поможет в создании матриц и векторов. После выполнения этих простых операций вся модель станет следующей:

$$\sum_{m} \lambda_i + \delta_i \rightarrow min, \text{для всех } i. \tag{50}$$

При условии:

$$C_i \sum_{m} W_{im} - \lambda_i + \delta_i = R_i - C_i, \text{для всех } i, \tag{51}$$

$$\sum_{m} Z_{im} = 1, \text{ для всех } i, \tag{52}$$

$$C_i \sum_{m} W_{im} \leq P_i U_i + (1 - P_i)[L_i + q(U_i - L_i)] - C_i, \text{для всех } i, \tag{53}$$

$$C_i \sum_{m} -W_{im} \leq C_i - L_i, \text{для всех } i, \tag{54}$$

$$-X_1 \leq -a_1, X_m \leq a_2, X_{M-1} - X_M \geq -a_3, \text{для всех } m \geq 2, \tag{55}$$

$$\sum_{m} mZ_{im} \leq e_i, \text{для всех } i, \tag{56}$$

$$\sum_{m} -mZ_{im} \leq d - e_i, \text{для всех } i, \tag{57}$$

$$\sum_{m} Z_{im} = 1, \text{для всех } i, \tag{58}$$

$$W_{im} - Z_{im} \leq 0, \text{для всех } i \text{ и } m, \tag{59}$$

$$X_m + Z_{im} - W_{im} \leq 1, \text{для всех } i \text{ и } m, \tag{60}$$

$$W_{im} - X_m \leq 0, \text{для всех } i \text{ и } m, \quad (61)$$

$$Z_{im} = 0{,}1, \text{для всех } i \text{ и } m. \quad (62)$$

Благодаря выведенным параметрам сгенерируем частично целочисленную задачу линейного программирования для решения описанной модели [21]. Добавим параметры, описанные выше, в написанный ранее программный код и произведём замеры времени у каждого алгоритма. Следуя из ранее сделанных выводов во разделе 3, для измерения времени работы алгоритмов будут использоваться только параллельные реализации [19].

Некоторые параметры, например, q, d, A и B, должны быть определены в дополнение к информации, представленными исходными данными. Эти параметры сильно влияют на размер поставленной задачи. Например, A и B устанавливают верхнюю и нижнюю границу интервалов для повышения заработной платы. Сложность и размер проблемы сильно зависят от разницы между этими верхними и нижними границами зарплаты. Кроме того, параметр d создает еще одну сложность, позволяя одному сотруднику быть кандидатом для нескольких категорий повышения заработной платы. Если предполагается, что d равно 1, то новая категория для повышения зарплаты, назначенная отделом кадров, должна быть либо такой же, как ранее определенная категория повышения, либо ниже. Таким образом, если принять d как большее число, то это позволит расширить границы данного процесса выбора и улучшит потенциал разработанной программы [22].

С другой стороны, пользователи (например, отдел кадров) программы могу свободно изменять числа для этих параметров из-за того, что эти параметры являются для них универсальным инструментом для управления и повышения заработной платы своим сотрудникам. Например, параметр q определяет верхнюю границу минимальной заработной платы некоторых сотрудников, что означает, что они могу иметь максимальную зарплату исходя из своего потенциала на работе. Таким образом, сбалансированная установка параметра q является важной частью этической оценки.

В таблице 24 показаны значения, используемые для параметров модели, описанной выше.

Таблица 24 – Значения параметров

Параметры	Значения
N	10
M	5
q	0,9
d	1
a_1	0,05
a_2	0,2
a_3	0,03
A	8
B	12
T	12
n	14

В связи с тем, что в данной модели оцениваются десять сотрудников с пятью различными категориями повышения заработной платы, N и M обозначены как 10 и 5 соответственно. В примере параметр q предполагается равным 0,9, что считается достаточно справедливым наказанием для сотрудников с низким потенциалом. Параметр d определяется как 1, что считается приемлемым как для самих сотрудников, так и для компании. Параметр a_1 – это самый низкий процент, который будет отдаваться сотрудникам с худшей производительностью [24]. В данном примере предполагается, что это 5% сотрудников, что считается минимальной величиной процента повышения зарплаты, позволяющей избежать увольнения с работы для низкоэффективных сотрудников. В тоже время самый высокий процент заработной платы a_2 установлен на уровне 20%. Разница между категориями повышения зарплаты a_3 в этом примере определяется как 3%. Данный проценты, безусловно, могут отличаться от отрасли к отрасли.

В качестве нижней границы повышения заработной платы обозначен, как интервал в 8 месяцев, а верхняя граница в 12 месяцев. Кроме того, весь временной интервал для периода повышения зарплаты также указан как 12 месяцев.

4.2 Применение алгоритма ветвей и границ полученной модели

Как упоминалось ранее, метод ветвей и границ предполагает построения узлов, дерева. На каждом узле (подзадаче) решаются с помощью целочисленного линейного программирования. Такие задачи решаются симплекс методом [30]. Также при работе написанного приложения засекается время работы алгоритма и идет подсчет количества узлов. В таблице 25 представлены результаты работы приложения.

Таблица 25 – Результаты работы приложения

Количество сотрудников	Время работы (секунды)	Количество узлов
10	3,754	202
15	4,442	1,524
20	20,013	10,186
25	175,534	923,649
30	937,598	4,259,723

Как видно из таблицы 25, время вычислений значительно увеличивается с увеличением числа сотрудников. Кроме того, другие элементы моделей также могут повлиять на время вычисления. Заявки с большим количеством сотрудников не выполняются из-за неэффективности используемого метода. Также это видно на графике, который был построен на основе таблицы 25 (рисунок 33).

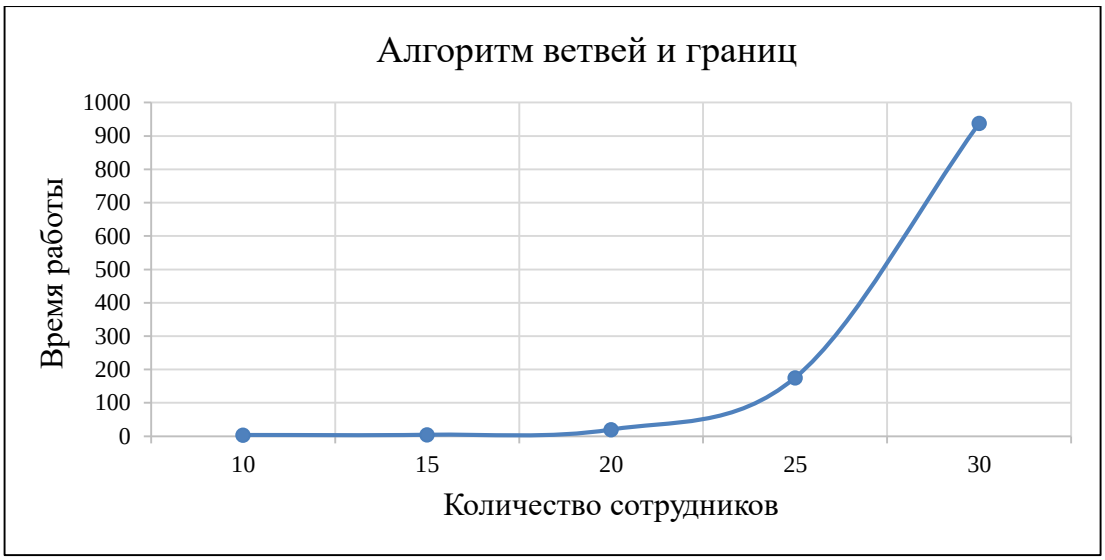

Рисунок 33 – График алгоритма ветвей и границ

4.3 Применение алгоритма Гомори полученной модели

Аналогично алгоритму ветвей и границ, используем данную модель на алгоритме Гомори. Следует также отметить, что включение в алгоритм метода Гомори увеличила скорость вычислений, уменьшив время выполнения с 3,754 секунд до 2,834 секунд. Время вычислений для различных случаев также представлено в таблице 26.

Таблица 26 – Результаты работы алгоритма Гомори

Количество сотрудников	Время работы (секунды)	Количество узлов
10	2,834	-
15	3,272	5
20	3,324	2
25	3,261	3
30	3,011	1
50	7,362	4
100	37,310	8

Таблица 4.4 доказывает, что включение метода Гомори в процедуру решения задачи значительно сокращает время вычислений и дает

пользователям возможность оценивать большее количество сотрудников. Построим график на основе таблицы 26 (рисунок 34).

Рисунок 34 – График алгоритма Гомори

4.4 Сравнительный анализ алгоритмов

Объединим таблицы 4.3 и 4.4 в одну, чтобы наглядно увидеть разницу во времени работы. Но так как в алгоритме Гомори использовалось большее количество сотрудников, то мы в общей таблице сократим это количество до количества в алгоритме ветвей и границ (таблица 27). Сравнение результатов из раздела 3 не совсем корректно, так как там использовались сухие «синтетические» данные.

Таблица 27 – Объединённая таблица двух алгоритмов

Количество сотрудников	Время работы алгоритма ветвей и границ (секунды)	Время работы алгоритма Гомори (секунды)
10	3,754	2,834
15	4,442	3,272
20	20,013	3,324

Продолжение таблицы 27

Количество сотрудников	Время работы алгоритма ветвей и границ (секунды)	Время работы алгоритма Гомори (секунды)
25	175,534	3,261
30	937,598	3,011

Для наглядности построим график на основе таблицы 27 (рисунок 35).

Рисунок 35 – Сравнение времени работы двух алгоритмов

В этом разделе подробно объясняется математическая модель, которая может применяться во многих фирмах на основе собственных предложений. Пример приложения выполняется с использованием подготовленного кода на языке программирования Python.

Благодаря сравнительному анализу наглядно видно, что в разработанной модели метод Гомори намного превосходит метод ветвей и границ по времени работы программного кода.

Заключение

В ходе выполнения магистерской диссертационной работы были решены следующие задачи:

- исследованы методы решения целочисленного и частично целочисленного задач линейного программирования;
- реализован однопоточный алгоритм ветвей и границ для целочисленных и частично-целочисленных задач линейного программирования на языке программирования Python;
- реализован параллельный алгоритм ветвей и границ для целочисленных и частично-целочисленных задач линейного программирования на языке программирования Python;
- реализован однопоточный алгоритм Гомори для целочисленных и частично-целочисленных задач линейного программирования на языке программирования Python;
- реализован параллельный алгоритм Гомори для целочисленных и частично-целочисленных задач линейного программирования на языке программирования Python;
- разработан пользовательский графический интерфейс.
- проведен сравнительный анализ на модели приближенной к реальным данным.

При помощи сравнительного анализа было выявлено, что на сухих данных и с использованием модели, предложенной Т. Кокиасара, стоит выбирать метод Гомори, так как он намного превосходит метод ветвей и границ по времени работы программного кода.

Список используемой литературы и используемых источников

1. Богданова Е.Л. Оптимизация в проектном менеджменте: линейное программирование: учебное пособие / Е.Л. Богданова, К.А. Соловейчик, К.Г. Аркина. – СПб.: Университет ИТМО, 2017. – 165 с.

2. Метод ветвей и границ [Электронный ресурс:] – Режим доступа: http://www.math.nsc.ru/AP/benchmarks/UFLP/uflp_bb.html

3. Решение задачи целочисленного программирования графическим методом и методом Гомори [Электронный ресурс:] – Режим доступа: https://www.matburo.ru/Examples/Files/LP_Num7.pdf

4. Решение задачи целочисленного программирования методом ветвей и границ [Электронный ресурс:] – Режим доступа: https://www.matburo.ru /Examples/Files/LP_Num5.pdf

5. Сизова С.А. Линейное программирование как область математического программирования при решении экономических задач / С.А. Сизова, В.Ю. Мурдугова, С.В. Мелешко // Старвопольский Государственный аграрный университет, статья в журнале – научная статья, №6(2), 2013, 16-20 с.

6. Шевченко В. Н. Линейное и целочисленное линейное программирование / В. Н. Шевченко, Н. Ю. Золотых, 2009, 154 с.

7. Экономико-математические методы [Электронный ресурс:] – Режим доступа: http://www.math.mrsu.ru/text/courses/method/post_zad1.htm

8. Юхименко Б.И. Модификации метода ветвей и границ для решения задач целочисленного линейного программирования и их эффекктивность / Б.И. Юхименко. – Украина, Інформатика та математичні методи в моделюванні, Том 5, №1, 2015. – 84-91 с.

9. Algorithms for Integer Programming [Электронный ресурс:] – Режим доступа: http://groups.di.unipi.it/optimize/Courses/RO2IG/aa1415/IP_algorithms.pdf

10. Approximation Algorithms using ILP [Электронный ресурс:] – Режим доступа: https://www.isical.ac.in/~arijit/courses/autumn2016/ILP-Approximation-Lecture-1.pdf

11. Branch and cut [Электронный ресурс:] – Режим доступа: https://en.wikipedia.org/wiki/Branch_and_cut

12. Bruno, J.E., Using linear programming salary evaluation models in collective bargaining negotiations with teacher unions. Socio Economic Planning Sciences // 1969, №3 (2). p. 103-117.

13. Castillo E. Mixed-Integer Linear Programming // E. Castillo, A.J. Conejo, P. Pedregal, R. Garcıa, N. Alguacil. – John Wiley and Sons, Inc, 2002.

14. Cokyasar T. A mixed integer linear programming approach for developing salary administration systems // University of Tennessee. – 2016. Taner Cokyasar

15. Cutting Plane Method [Электронный ресурс:] – Режим доступа: https://www.math.cuhk.edu.hk/course_builder/1415/math3220/L5.pdf

16. Fabozzi, F.J. and A.W. Bachner, Mathematical programming models to determine civil service salaries // European Journal of Operational Research, 1979. №3(3). p. 190-198.

17. Integer Programming the Branch and Bound method [Электронный ресурс:] – Режим доступа: http://web.tecnico.ulisboa.pt/mcasquilho/compute/_linpro/TaylorB_module_c.pdf

18. József Abaffy. Solving Integer and Mixed Integer Linear Problems with ABS Method // Óbuda University, 2016.

19. Land, A.H. An automatic of solving discreate programming problems / A.H. Land, A.G. Doig // Econometrica. 1960. Vol. 28, №3. p. 497-520.

20. Larrosa J. Mixed integer linear programming // J. Larrosa, A. Oliveras, E. Rodrıguez-Carbonell, April, 2009.

21. McCarl, B.A., Applied mathematical programming using algebraic systems/ B.A. McCarl, T.H. Spreen // Cambridge, MA, 1997.

22. Mixed Integer Linear Programming [Электронный ресурс:] – Режим доступа: https://www.cs.upc.edu/~erodri/webpage/cps/theory/lp/milp/slides.pdf

23. Mixed-Integer Programming (MIP) – A Primer on the Basics [Электронный ресурс:] – Режим доступа: https://www.gurobi.com/resource/mip-basics/

24. Robert E. Bixby. A brief history of linear and mixed-integer programming computation // Extra Volume ISMP. – 2012. p. 107–121.

25. So Han Florence Yip. A Mixed-Integer Linear Programming Model for Disaster Housing Supply Chain Design and Optimization // Mechanical Engineering, University of Rochester. – 2017.

26. Survey A. Linear Integer Programming Methods and Approaches // Bulgarian Academy of sciences. – 2016.

27. Traveling-salesman-problem [Электронный ресурс:] – Режим доступа: http://examples.gurobi.com/traveling-salesman-problem/#demo

28. Winston, W.L. Operations research: applications and algorithms // Duxbury press Belmont, CA, №3. 2014

29. Williams, H Paul. Model Building in Mathematical Programming // John Wiley & Sons, 2013.

30. Xioxia L. Mixed integer linear programming in process scheduling: modeling, algorithms, and applications // Springer Science. – 2005.

31. Zhang X. P., Restructured electric power systems: analysis of electricity markets with equilibrium models // John Wiley & Sons, 2016, vol. 71.

Приложение А

Листинг программы

```
import math
import time
from threading import Thread

from api.Simplex import SimplexMethodSolver as simplex


class BranchAndBorderMethod:
    """
    Метод ветвей и границ
    """

    @staticmethod
    def find_solution_with_all_params(function,
bounds_array, optimal_model, is_multiple_threads=False,
expect_nulls=False, max_depth=10):
        return BranchAndBorderMethod \
            .__get_solution(function, bounds_array,
optimal_model, is_multiple_threads, expect_nulls,
max_depth)

    @staticmethod
    def find_solution_with_time(function, bounds_array,
optimal_model, is_multiple_threads=False,
expect_nulls=False, max_depth=10):
        result, work_time, depth =
BranchAndBorderMethod \
            .__get_solution(function, bounds_array,
optimal_model, is_multiple_threads, expect_nulls,
max_depth)
        return result, work_time

    @staticmethod
    def find_solution(function, bounds_array,
optimal_model, is_multiple_threads=False,
expect_nulls=False, max_depth=10):
        return BranchAndBorderMethod\
            .__get_solution(function, bounds_array,
optimal_model, is_multiple_threads, expect_nulls,
max_depth)[0]
```

```
    @staticmethod
    def __get_solution(function, bounds_array,
optimal_model, is_multiple_threads=False,
expect_nulls=False, max_depth=10):
        """
        Поиск решения
        :param expect_nulls: Параметры который говорит,
ожидаем ли мы нулевые значения
        :param max_depth: Максимальное количество слоёв
дерева
        :param function: Целевая функция
        :param bounds_array: Ограничения
        :param optimal_model: Оптимальная модель
        :param is_multiple_threads: Использование
нескольких потоков
        :return:
        """
        if is_multiple_threads is False:
            return BranchAndBorderMethod \

.__find_solution_by_single_thread(function,
bounds_array, optimal_model, max_depth, expect_nulls)
        else:
            return BranchAndBorderMethod\

.__find_solution_by_multiple_threads(function,
bounds_array, optimal_model, max_depth, expect_nulls)

    @staticmethod
    def __find_solution_by_multiple_threads(function,
bounds_array, optimal_model, max_depth, expect_nulls):
        """
        Поиск решения в несколько потоков
        :param function: Целевая функция
        :param bounds_array: Ограничения
        :param optimal_model: Оптимальная модель
        :param max_depth: Максимальная глубина дерева
        :param expect_nulls: Ожидание нулевых значений
        :return: Результат, коэффициенты xn, время
работы программы, максимальная глубина дерева
        """
        start_timer = time.time()
        tree = Tree(bounds_array, function,
optimal_model)  # Создаём древовидную структуру
```

```
        main_thread = NewThread(tree.root, max_depth)
        main_thread.start()
        main_thread.join()
        tree = main_thread.get_result()

        return BranchAndBorderMethod.__find_max(tree,
expect_nulls) \
            if optimal_model else
BranchAndBorderMethod.__find_min(tree, expect_nulls), \
            time.time() - start_timer,
BranchAndBorderMethod.__get_max_depth(tree)

    @staticmethod
    def __find_solution_by_single_thread(function,
bounds_array, optimal_model, max_depth, expect_nulls):
        """
        Поиск решения в одиночном потоке
        :param function: Целевая функция
        :param bounds_array: Ограничения
        :param optimal_model: Оптимальная модель
        :param max_depth: Максимальная глубина дерева
        :param expect_nulls: Ожидание нулевых значений
        :return: Результат, коэффициенты xn, время
работы программы, максимальная глубина дерева
        """
        start_timer = time.time()
        tree = Tree(bounds_array, function,
optimal_model)  # Создаём древовидную структуру
        tree =
BranchAndBorderMethod.__recount_node(tree.root,
max_depth)  # Строим дерево

        return BranchAndBorderMethod.__find_max(tree,
expect_nulls) \
            if optimal_model else
BranchAndBorderMethod.__find_min(tree, expect_nulls), \
            time.time() - start_timer,
BranchAndBorderMethod.__get_max_depth(tree)

    @staticmethod
    def __recount_node(node, max_depth):
        """
        Пересчитать значение в узле
        :param max_depth: Максимальная глубина дерева
        :param node: Узел дерева
```

```
        :return: Дерево решений
        """
        # Ищем решение
        result, variables =
simplex.find_solution(node.get_function(),
node.get_bounds(), node.get_model())
        node.set_results(result, variables)  # Задаём
текущему узлу получившиеся значения

        float_value, float_num =
BranchAndBorderMethod.get_not_integer_var(variables)  #
Находим дробное значение

        if node.level < max_depth and result is not
None:
            if float_num != -1:
                # Создаём доп ограничения
                left_bound, right_bound =
BranchAndBorderMethod.create_new_bounds(float_value,
float_num, node)

                if left_bound is not None:
                    # Создаём левый узел с
дополнительным ограничением
                    node.set_left(node.get_bounds(),
left_bound, node.get_function(), node.get_model(),
node.level + 1)

BranchAndBorderMethod.__recount_node(node.get_left(),
max_depth)

                if right_bound is not None:
                    # Создаём правый узел с
дополнительным ограничением
                    node.set_right(node.get_bounds(),
right_bound, node.get_function(), node.get_model(),
node.level + 1)

BranchAndBorderMethod.__recount_node(node.get_right(),
max_depth)

        return node

    @staticmethod
    def get_not_integer_var(variables):
```

```
        """
        Поиск не целого числа из списка всех полученных
значений xn
        :param variables: Массив значений xn
        :return: Номер икса, имеющего дробный
коэффициент
        """
        for i in range(len(variables)):
            if variables[i] > 0 and not
variables[i].is_integer():
                return variables[i], (i + 1)
        return -1, -1

    @staticmethod
    def __is_contains_nulls(variables):
        """
        Содержит ли массив нулевые значения
        :param variables: Массив значений
        :return: True - содержит, False - не содержит
        """
        return sum([1.0 if v == 0 else 0.0 for v in
variables]) > 0

    @staticmethod
    def create_new_bounds(value, num_of_x, node):
        """
        Создание дополнительных ограничений
        :param num_of_x: Номер x
        :param value: Значение коэффициента
        :return: Границы для левого и правого узла
дерева
        """
        int_val = math.modf(value)[1]
        left_bound = str("x" + str(num_of_x) + ">=" +
str(int_val + 1))
        right_bound = str("x" + str(num_of_x) + "<=" +
str(int_val))

        return
BranchAndBorderMethod.__get_not_equal_bound(left_bound,
node.bound_array), \

BranchAndBorderMethod.__get_not_equal_bound(right_bound
, node.bound_array)
```

```
    @staticmethod
    def __get_not_equal_bound(bound, bounds_array):
        """
        Вернуть уникальную границу
        :param bound: Новая граница
        :param bounds_array: Существующие границы
        :return: bound, если она уникальна, иначе None
        """
        if bounds_array.count(bound) > 0:
            return None

        sub_str = bound[:bound.find("=") - 1]
        for b in bounds_array:
            if (sub_str + ">=") in b or (sub_str +
"<=") in b:
                return None
        return bound

    @staticmethod
    def __get_max_depth(root):
        """
        Поиск высоты дерева
        :param root: Корневой узел
        :return: Высота дерева
        """
        nods_array =
BranchAndBorderMethod.__get_all_nodes(root)
        max_depth = 0

        for node in nods_array:
            if node.level > max_depth:
                max_depth = node.level
        return max_depth

    @staticmethod
    def __find_min(root, expect_nulls):
        """
        Поиск наименьшего решения
        :param expect_nulls: Параметр ожидания нулевых
значений
        :param root: Корневой узел
        :return: Наименьшее решение
        """
```

```
        nods_array =
BranchAndBorderMethod.__get_all_results(root,
expect_nulls)
        min_value = float("inf")
        variables = None

        for node in nods_array:
            if node.result is not None and node.result
< min_value:
                min_value = node.result
                variables = node.variables

        return min_value, variables

    @staticmethod
    def __find_max(root, expect_nulls):
        """
        Поиск наибольшего решения
        :param expect_nulls: Параметр ожидания нулевых
значений
        :param root: Корневой узел
        :return: Наибольшее решение
        """
        nods_array =
BranchAndBorderMethod.__get_all_results(root,
expect_nulls)
        min_value = float("-inf")
        variables = None

        for node in nods_array:
            if node.result is not None and node.result
> min_value:
                min_value = node.result
                variables = node.variables

        return min_value, variables

    @staticmethod
    def __get_all_nodes(node, result=None):
        """
        Получение всех узлов
        :param node: Узел
        :param result: Массив всех узлов
        :return: Массив всех узлов
        """
```

```
        result = [] if result is None else result
        result.append(node)

        if node.get_left() is not None:

BranchAndBorderMethod.__get_all_nodes(node.get_left(),
result)
        if node.get_right() is not None:

BranchAndBorderMethod.__get_all_nodes(node.get_right(),
result)

        return result

    @staticmethod
    def __get_all_results(node, expect_nulls,
result=None):
        """
        Получение всех узлов, которые содержат решение
        :param node: Узел
        :param result: Массив всех узлов решений
        :return: Массив всех узлов решений
        """
        result = [] if result is None else result
        if
BranchAndBorderMethod.get_not_integer_var(node.variable
s)[0] == -1 \
                and (not expect_nulls or not
BranchAndBorderMethod.__is_contains_nulls(node.variable
s)):
            result.append(node)

        if node.get_left() is not None:

BranchAndBorderMethod.__get_all_results(node.get_left()
, expect_nulls, result)
        if node.get_right() is not None:

BranchAndBorderMethod.__get_all_results(node.get_right(
), expect_nulls, result)

        return result


class NewThread(Thread):
```

```
    """
    Поток выполнения
    """
    def __init__(self, node, max_depth):
        Thread.__init__(self)
        self.name = "Thread [Level = " +
str(node.level) + ", borders = " +
str(node.bound_array) + "]"
        self.node = node
        self.max_depth = max_depth

    def run(self):
        """
        Функция расчёта значения узла
        """

        # Ищем решение
        result, variables =
simplex.find_solution(self.node.get_function(),
self.node.get_bounds(), self.node.get_model())
        self.node.set_results(result, variables)  #
Задаём текущему узлу получившиеся значения

        float_value, float_num =
BranchAndBorderMethod.get_not_integer_var(variables)  #
Находим дробное значение

        if self.node.level < self.max_depth and result
is not None:
            if float_num != -1:
                # Создаём доп ограничения
                left_bound, right_bound =
BranchAndBorderMethod.create_new_bounds(float_value,
float_num, self.node)

                if left_bound is not None:
                    # Создаём левый узел с
дополнительным ограничением

self.node.set_left(self.node.get_bounds(), left_bound,
self.node.get_function(),

self.node.get_model(), self.node.level + 1)
```

```
                    left_thread =
NewThread(self.node.get_left(), self.max_depth)
                    left_thread.start()
                    left_thread.join()

                if right_bound is not None:
                    # Создаём правый узел с
дополнительным ограничением

self.node.set_right(self.node.get_bounds(),
right_bound, self.node.get_function(),

self.node.get_model(), self.node.level + 1)

                    right_thread =
NewThread(self.node.get_right(), self.max_depth)
                    right_thread.start()
                    right_thread.join()

    def get_result(self):
        return self.node


class Tree:
    def __init__(self, bound_array, function,
optimal_model):
        self.root = Node(None, None, bound_array,
function, optimal_model, 1)


class Node:
    def __init__(self, result, variables, bound_array,
function, optimal_model, level=None, left=None,
right=None):
        self.result = result  # Значение целевой
функции
        self.variables = variables  # Значение всех
переменных (x1...xn)
        self.bound_array = bound_array  # Ограничения
на данном узле
        self.function = function  # Целевая функция
        self.optimal_model = optimal_model  #
Оптимальная модель MAX/MIN
        self.level = level  # Уровень дерева
        self.left = left
```

```
        self.right = right

    def set_results(self, result, variables):
        self.result = result
        self.variables = variables

    def set_left(self, bound_array, new_bound,
function, optimal_model, level):
        self.left = Node(None, None, bound_array +
[new_bound], function, optimal_model, level)

    def set_right(self, bound_array, new_bound,
function, optimal_model, level):
        self.right = Node(None, None, bound_array +
[new_bound], function, optimal_model, level)

    def get_left(self):
        return self.left

    def get_right(self):
        return self.right

    def get_bounds(self):
        return self.bound_array

    def get_function(self):
        return self.function

    def get_model(self):
        return self.optimal_model

    def __str__(self):
        return "Node [result = " + str(self.result) +
", vars = " + str(self.variables) \
               + ", level = " + str(self.level) \
               + ", borders = " + str(self.bound_array)
\
               + "]"

import random


class GeneratorUtils:

    @staticmethod
```

```
    def generate_liner_condition(num_of_args,
left_border=-100, right_border=100):
        """
        Генерация ЦЛП задачи
        :param num_of_args: Количество аргументов
        :param left_border: Левая граница генерируемых
чисел
        :param right_border: Правая граница
генерируемых чисел
        :return: Целевая функция, ограничения
        """
        function =
GeneratorUtils.__generate_left_part(num_of_args,
left_border, right_border)
        borders = []

        for _ in range(num_of_args):

borders.append(GeneratorUtils.__generate_left_part(num_
of_args, left_border, right_border) +

GeneratorUtils.__generate_right_part(left_border,
right_border))

        return function, borders

    @staticmethod
    def __generate_left_part(num_of_args, left_border,
right_border):
        """
        Генерация левой части уравнения
        :param num_of_args: Количество аргументов
        :param left_border: Левая граница генерируемых
чисел
        :param right_border: Правая граница
генерируемых чисел
        :return: Левая часть уравнения
        """
        function = ""
        for arg_num in range(num_of_args):
            fun_cof = random.randint(left_border,
right_border)
            function += ((str(fun_cof) if fun_cof < 0
else ("+" + str(fun_cof))) + "x" + str(arg_num + 1))
```

```
        return function

    @staticmethod
    def __generate_right_part(left_border,
right_border):
        """
        Генерация правой части уравнения
        :param left_border: Левая граница генерируемых
чисел
        :param right_border: Правая граница
генерируемых чисел
        :return: Правая часть уравнения
        """
        fun_cof = random.randint(left_border,
right_border)
        return ("<=" if fun_cof > 0 else ">=") +
str(abs(fun_cof))


import re


class Parser:

    # Константы границ
    LESS_OR_EQUAL = "<="
    EQUAL = "="
    MORE_OR_EQUAL = ">="

    # Лист всех границ
    __BORDERS = [LESS_OR_EQUAL, MORE_OR_EQUAL, EQUAL]

    # Мар для всех символов и их порядкого номера в КА
    __SYMBOLS = {
        "+": 0, "-": 0, "0": 1, "1": 1, "2": 1,
        "3": 1, "4": 1, "5": 1, "6": 1, "7": 1,
        "8": 1, "9": 1, ".": 2, ",": 2, "*": 3,
        "x": 4, None: 5
    }

    # Расшифровки к ошибкам:
    # US — Unexpected symbol (Неожиданный символ)
    # UF — Unfinished formula (Незаконченная формула)
    # UC - Unknown character (Неизвестный символ)
```

```
    # BCE — No boundary conditions (Отсутсвуют
граничные условия)
    __EXCEPTIONS = {
        "US": "Неожиданный символ '{}'!",
        "UF": "Формула не закночена!",
        "UC": "Неизвестный символ '{}'!",
        "BCE": "Отсутсвуют граничные условия или
неверный разделитель!"
    }

    # Переменная окончания парсинга
    # EP — End Of Parsing (Конец парсинга: выход из
программы)
    __END_OF_PARSING = "EP"

    # Таблица состояний (КА)
    __STATES = [
        [1, 2, 'US', 'US', 5, 'UF'],        # 0 — Вход
в конечный автомат
        ['US', 2, 'US', 'US', 5, 'UF'],     # 1 — Знак
перед числом (+/-)
        ['US', 2, 3, 4, 5, 'EP'],           # 2 — Ввод
числа
        ['US', 2, 'US', 'US', 'US', 'UF'],  # 3 — Ввод
точки в числе перед иксом
        ['US', 'US', 'US', 'US', 5, 'UF'],  # 4 — Ввод
умножения
        ['US', 6, 'US', 'US', 'US', 'UF'],  # 5 — Ввод
икса
        [0, 6, 'US', 'US', 'US', 'EP']      # 6 — Ввод
номера икса
    ]

    @staticmethod
    def parse_function_with_borders(str_func,
const_count):
        """
        Парсить функию с граничными условиями
                Пример: 2*x1 + 3*x2 >= 100
                Пример: 2*x1 + 3*x2 = 4*x3
        :param str_func: Функция в виде строки
        :param const_count: Количество используемых
констант
        :return: Массив векторов коэффициентов для
левой и правой части,
```

```
        массив свободных членов для левой и правой
части
        """
        str_func = str_func.replace(" ", "")

        # Проверяем, есть ли граничные условия
        if not Parser.__is_contains_borders(str_func):
            raise
ParserException(Parser.__EXCEPTIONS["ВСЕ"])

        # Задаём левые и правые части формулы
        left_side, right_side, border =
Parser.__split_formula(str_func)

        left_cof, left_free_cof =
Parser.__parse_side_of_function(left_side, const_count)
        right_cof, right_free_cof =
Parser.__parse_side_of_function(right_side,
const_count)

        return [left_cof, right_cof], [left_free_cof,
right_free_cof], border

    @staticmethod
    def parse_alone_function(str_func, const_count):
        """
        Парсить функию без граничных условий
                Пример: 2*x1 + 3*x2
                Пример: 2*x1 + 3*x2 - 4*x3
        :param str_func: Функция в виде строки
        :param const_count: Количество используемых
констант
        :return: Вектор коэффициентов, свободный член
        """
        str_func = str_func.replace(" ", "")
        fun_cof, fun_free =
Parser.__parse_side_of_function(str_func, const_count)
        return fun_cof

    @staticmethod
    def find_max_constant(func_array):
        """
        Парсит целевую функцию и все ограничения, для
нахождения
        наибольшего номера X
```

```
        :param func_array: Массив с целевой функцией и
всех ограничений
        :return: Номер наибольшего X
        """
        pattern = "x\\d+"
        max_num = 0

        for str_func in func_array:
            for num in re.findall(pattern, str_func):
                if int(num[1:]) > max_num:
                    max_num = int(num[1:])

        return max_num

    @staticmethod
    def __parse_side_of_function(str_func,
const_count):
        """
        Парсинг одной части функции
        :param str_func: Часть функции в виде строки
        :param const_count: Количество используемых
констант
        :return: Вектор коэффициентов, свободный член
        """
        before_state = 0  # Предыдущее состояние
        new_state = 0  # Следующее состояние

        cof_array = [0 for _ in range(const_count)]  #
Массив коэффициентов
        free_cof = None  # Свободный коэффициент

        buffer = None  # Буффер для значений
        buffered_cof = None  # Последний сохранённый
коэффициент

        for i in range(len(str_func)):  # Читаем строку
посимвольно
            current_char = str_func[i]  # Берём текущую
строку

            if
Parser.__SYMBOLS.__contains__(current_char):  # Если в
таблице символов данный символ поддерживается
```

```
                new_state =
Parser.__STATES[before_state][Parser.__SYMBOLS[current_
char]]  # Берём новое состояние

                if new_state == "US":  # Если мы
перешли в неправильное состояние, выдаём ошибку
                    raise
ParserException(Parser.__EXCEPTIONS["US"].format(curren
t_char))

                # Если нужно запомнить текущий символ в
буффер
                if
Parser.__is_need_to_save_char(before_state, new_state):
                    buffer = current_char if buffer is
None else (buffer + current_char)

                # Если нужно записать коэффициент в
буффер
                elif
Parser.__is_need_update_cof_buffer(before_state,
new_state):
                    buffered_cof =
Parser.__resolve_to_float(buffer)
                    buffer = None

                # Если нужно записать коэффициент в
массив
                elif
Parser.__is_needed_to_save_cof(before_state,
new_state):
                    cof_array[int(buffer) - 1] =
buffered_cof
                    buffered_cof = None
                    buffer = current_char
            else:
                raise
ParserException(Parser.__EXCEPTIONS["UC"].format(curren
t_char))  # Иначе вызываем ошибку

            before_state = new_state  # Обновляем
предыдущее состояние КА

        if len(buffer) != 0 and buffered_cof is not
None:
```

```
            cof_array[int(buffer) - 1] = buffered_cof
        elif len(buffer) != 0 and buffered_cof is None:
            try:
                free_cof = float(buffer)
            except ValueError:
                raise
ParserException(Parser.__EXCEPTIONS["UF"])  # Формула
не закончена, вызываем ошибку

        if
Parser.__STATES[before_state][Parser.__SYMBOLS[None]]
== Parser.__END_OF_PARSING:  # Если мы вышли из КА
            return cof_array, free_cof
        else:
            raise
ParserException(Parser.__EXCEPTIONS["UF"])  # Формула
не закончена, вызываем ошибку

    @staticmethod
    def __is_needed_to_save_cof(old_state, new_state):
        """
        Проверка на необходимость обнулить буфер
        :param old_state:
        :param new_state:
        :return:
        """
        if old_state == 6 and new_state == 0:
            return True
        return False

    @staticmethod
    def __is_need_update_cof_buffer(old_state,
new_state):
        """
        Проверка на необходимость записать значение из
буфера в буфер коэфициентов
        :param old_state: Предыдущее состояние
        :param new_state: Новое состояние
        :return: True — обновить буфер / False — не
обновлять буфер
        """
        if (old_state == 2 and (new_state == 4 or
new_state == 5)) or (old_state < 2 and new_state == 5):
            return True
        return False
```

```
    @staticmethod
    def __is_need_to_save_char(old_state, new_state):
        """
        Проверка на необходимость записать текущий
символ в буффер
        * Это происходит только при запоминании чисел
        :param old_state: Предыдущее состояние
        :param new_state: Новое состояние
        :return: True — записать символ в буффер /
False — не записывать число в буффер
        """
        if old_state == new_state \
                or old_state < 4 and new_state < 4 \
                or old_state == 5 and new_state == 6:
            return True
        return False

    @staticmethod
    def __is_contains_borders(str_func):
        """
        Проверочный метод, содержит ли строка граничный
знак
        :param str_func: Функция в виде строки
        :return: True — строка содержит граничный знак
/ False — не содержит
        """
        for border in Parser.__BORDERS:
            if border in str_func:
                return True
        return False

    @staticmethod
    def __split_formula(str_func):
        """
        Делит строку по граничному знаку ('>', '<',
'>=', '<=', '=')
        :param str_func: Функция в виде строки
        :return: Левая часть функции, правая часть
функции, разделяющий знак
        """
        for border in Parser.__BORDERS:
            if border in str_func:
                return str_func.split(border)[0],
str_func.split(border)[1], border
```

```
    @staticmethod
    def __resolve_to_float(str_num):
        """
        Перевод строки в число
        Если на вход придёт строка, содержащая только
знак "+" или "-",
        то вернуться соотвественно "+1" и "-1"
        :param str_num: Число в виде строки
        :return: Число float
        """
        if str_num is None:
            return 1
        if str_num == "-":
            return -1
        elif str_num == "+":
            return 1
        else:
            return float(str_num)


class ParserException(Exception):
    def __init__(self, message):
        self.message = message
        Exception.__init__(self, message)

    def get_message(self):
        return self.message


from pulp import LpVariable, LpProblem, LpMaximize,
LpMinimize, value
from api.Parser import Parser


class SimplexMethodSolver:
    """
    Класс для решения симплекс-методом
    """

    @staticmethod
    def find_solution(function, bounds_array,
optimal_model):
        """
        Поиск решения симплек-методом
```

```
        :param optimal_model: True — MAX / False — MIN
        :param function: Функция
        :param bounds_array: Условия
        :return: Решение симплекс-методом
        """
        max_num_of_func =
Parser.find_max_constant(function)
        max_num_of_bounds =
Parser.find_max_constant(bounds_array)
        num_of_const = max_num_of_func if
max_num_of_func > max_num_of_bounds else
max_num_of_bounds

        x = [LpVariable("x" + str(i + 1), lowBound=0)
for i in range(num_of_const)]
        problem = LpProblem("0", LpMaximize) if
optimal_model else LpProblem("0", LpMinimize)

        # Считаем main функцию
        fun_cof = Parser.parse_alone_function(function,
num_of_const)
        problem += sum([fun_cof[i] * x[i] for i in
range(num_of_const)])

        coefficients = []
        arguments = []
        borders = []

        # Считаем коэфициенты
        for bound in bounds_array:
            cof_arr, free_arr, border =
Parser.parse_function_with_borders(bound, num_of_const)

            # Запоминаем всю информацию о начальных
условиях для дальнейшей проверки
            coefficients.append(cof_arr)
            arguments.append(free_arr)
            borders.append(border)

            if border == Parser.MORE_OR_EQUAL:
                problem += \
                    sum([cof_arr[0][i] * x[i] for i in
range(num_of_const)]) + (
                        0 if free_arr[0] is None else
free_arr[0]) >= \
```

```
                    sum([cof_arr[1][i] * x[i] for i in
range(num_of_const)]) + (
                        0 if free_arr[1] is None else
free_arr[1])
            elif border == Parser.LESS_OR_EQUAL:
                problem += \
                    sum([cof_arr[0][i] * x[i] for i in
range(num_of_const)]) + (
                        0 if free_arr[0] is None else
free_arr[0]) <= \
                    sum([cof_arr[1][i] * x[i] for i in
range(num_of_const)]) + (
                        0 if free_arr[1] is None else
free_arr[1])
            elif border == Parser.EQUAL:
                problem += \
                    sum([cof_arr[0][i] * x[i] for i in
range(num_of_const)]) + (
                        0 if free_arr[0] is None else
free_arr[0]) == \
                    sum([cof_arr[1][i] * x[i] for i in
range(num_of_const)]) + (
                        0 if free_arr[1] is None else
free_arr[1])

        problem.solve()
        variables = [var.varValue for var in
problem.variables()]

        if
SimplexMethodSolver.__check_to_normal(coefficients,
arguments, borders, variables):
            return problem.objective.value(), variables
        return None, []

    @staticmethod
    def __check_to_normal(coefficients, arguments,
borders, variables):
        """
        Проверка на возможность решения
        :param coefficients: Массив коэффициентов
        :param arguments: Массив аргументов
        :param borders: Массив условий
        :param variables: Массив значений xn
```

```
        :return: True - Возможность существования /
False - Не возможность существования
        """
        for i in range(len(variables)):
            left = 0.0
            right = 0.0

            for j in range(len(variables)):
                left += float(coefficients[i][0][j]) *
float(variables[j])
            left += float(arguments[i][0]) if not
arguments[i][0] is None else 0.0

            for j in range(len(variables)):
                right += float(coefficients[i][1][j]) *
float(variables[j])
            right += float(arguments[i][1]) if not
arguments[i][1] is None else 0.0

            if abs(left - right) > 0.00001:
                if borders[i] == Parser.MORE_OR_EQUAL:
                    if not left >= right:
                        return False
                elif borders[i] ==
Parser.LESS_OR_EQUAL:
                    if not left <= right:
                        return False
                else:
                    if not left == right:
                        return False
        return True
```

Приложение Б
Листинг пользовательского интерфейса

```
import tkinter.ttk as ttk
from tkinter import *

from api.BranchAndBorderMethod import
BranchAndBorderMethod

limitations_text = []
limitations_entries = []
borders_labels = []

def number_of_limitations():
    global limitations_text
    global limitations_entries
    global borders_labels

    count_limitations_int = 0 if count_limitations_str
== '' else int(count_limitations_str.get())
    limitations_text = [StringVar() for _ in
range(count_limitations_int)]
    limitations_entries =
[Entry(textvariable=limitations_text[i]) for i in
range(count_limitations_int)]
    borders_labels = []
    for number in range(count_limitations_int):
        borders_labels.append(Label(text=str(number+1)
+ ") ", fg="#000000", bg=bg, font='Times 14'))
        borders_labels[number].grid(row=number + 2,
column=0, sticky="e")
```

```
        limitations_entries[number].grid(row=number+2,
column=1, padx=5, pady=5)


def BnP():
    global limitations_text
    global bnp_ordinary

    func = target_function_str.get()
    borders = [text.get() for text in limitations_text]
    optimal = (True if check.get() == 1 else False)

    result, timer =
BranchAndBorderMethod.find_solution_with_time(func,
borders, optimal)
    bnp_ordinary.entry["text"] = "Result = " +
str(result[0]) + ", vars = " + str(result[1]) + ", time
= " + str(timer)

def BnP_parall():
    global limitations_text
    global bnp_parallel

    func = target_function_str.get()

    borders = [text.get() for text in limitations_text]
    optimal = (True if check.get() == 1 else False)
```

```
    result, timer =
BranchAndBorderMethod.find_solution_with_time(func,
borders, optimal)
    bnp_parallel.entry["text"] = "Result = " +
str(result[0]) + ", vars = " + str(result[1]) + ", time
= " + str(timer)

#  цвет
bg = '#e7f2e1'
#  создание окна интерфейса
root = Tk()
root.configure(background=bg)
root.title("ЦЛП методом ветвей и границ")
root.geometry("1000x200")

s = ttk.Style()
s.configure('Wild.TRadiobutton', background=bg,
foreground='black')

target_function_str = StringVar()
target_function = Label(text="Введите целевую
функцию:", fg="#000000", bg=bg, font='Times 14')
target_function.grid(row=0, column=0, sticky="w")
# поле для записи целевой функции
target_function_entry =
Entry(textvariable=target_function_str)
target_function_entry.grid(row=0, column=1, padx=5,
pady=5)
```

```
# радио кнопка на мин или макс функции
check = IntVar()
max_radiobutton = ttk.Radiobutton(text="Max",
style='Wild.TRadiobutton', value=1, variable=check)
max_radiobutton.grid(row=0, column=2, sticky="w")
min_radiobutton = ttk.Radiobutton(text="Min",
style='Wild.TRadiobutton', value=2, variable=check)
min_radiobutton.grid(row=0, column=3, sticky="w")

count_limitations_str = StringVar()
count_limitations = Label(text="Количество
ограничений:", fg="#000000", bg=bg, font='Times 14')
count_limitations.grid(row=1, column=0, sticky="w")
count_limitations_entry =
Entry(textvariable=count_limitations_str)
count_limitations_entry.grid(row=1, column=1, padx=5,
pady=5)

entering_limitations = Button(text="Ок", fg="#000000",
bg="#c6f2ae", font='Times 14',
command=number_of_limitations)
entering_limitations.grid(row=1, column=2, sticky="w")

entering_limitations = Button(text="Рандом",
fg="#000000", bg="#c6f2ae", font='Times 14',
command=number_of_limitations)
entering_limitations.grid(row=1, column=3, sticky="w")

bnp_str = StringVar()
```

```
#  кнопка для решения цлп методом ветвей и границ
bnp_ordinary = Button(text="Метод ветей и границ (один
поток)", fg="#000000", bg="#c6f2ae", font='Times 14',
command=BnP)
bnp_ordinary.grid(row=0, column=4, padx=25, pady=5,
sticky="w")
bnp_ordinary.entry = Label(padx=0, pady=0, font='Times
14', bg=bg)
bnp_ordinary.entry.grid(row=0, column=5, padx=5,
pady=5)

#  кнопка для решения цлп методом ветвей и границ
параллельный
bnp_parallel = Button(text="Метод ветей и границ
(параллельный)", fg="#000000", bg="#c6f2ae",
font='Times 14', command=BnP_parall)
bnp_parallel.grid(row=1, column=4, padx=5, pady=5,
sticky="w")
bnp_parallel.entry = Label(padx=0, pady=0, font='Times
14', bg=bg)
bnp_parallel.entry.grid(row=1, column=5, padx=5,
pady=5)

#  Время работы алгоритма
time = Label(padx=0, pady=0, font='Times 14',
bg="#e7f2e1")
time.grid(row=1, column=0, sticky=W, columnspan=3)
#  Сумма построенного маршрута
summs = Label(padx=0, pady=0, font='Times 14', bg=bg)
summs.grid(row=2, column=0, sticky=W, columnspan=3)
```

```
#  Проверка на некорректный ввод
stub1 = Label(font='Times 14', bg=bg, fg='red')
stub1.grid(row=3, column=0, sticky="w", columnspan=3)
#  Проверка на некорректный ввод
stub2 = Label(font='Times 14', bg=bg, fg='red')
stub2.grid(row=4, column=0, sticky="w", columnspan=3)
#  Проверка на некорректный ввод
output = Label(font='Times 14', bg=bg, fg='red')
output.grid(row=5, column=0, sticky="w", columnspan=3)
root.mainloop()
```