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 кафедры)
01.03.02 Прикладная математика и информатика
(код и наименование направления подготовки, специальности)
Системное программирование и компьютерные технологии
(направленность (профиль)/специализация

БАКАЛАВРСКАЯ РАБОТА

на тему **Исследование различных методов решения задачи коммивояжёра**

Студент	Е.А. Кувшинова	
	(И.О. Фамилия)	(личная подпись)
Руководитель	М.А. Тренина	
	(И.О. Фамилия)	(личная подпись)
Консультанты	Н.В. Андрюхина	
	(И.О. Фамилия)	(личная подпись)

Допустить к защите

Заведующий кафедрой к.т.н., доцент, А.В. Очеповский _____
(ученая степень, звание, И.О. Фамилия) (личная подпись)
«_____»_____20_____г.

Тольятти 2019

АННОТАЦИЯ

Тема бакалаврской работы: «Исследование различных методов решения задачи коммивояжёра».

Работа выполнена студенткой Тольяттинского государственного университета, института математики, физики и информационных технологий, группы ПМИб-1502, Кувшиновой Евгенией Андреевной. Выпускная квалификационная работа посвящена исследованию методов решения задачи коммивояжера и разработке программного кода, реализующего их. В данной работе используются следующие методы: ветвей и границ и ближайшего соседа, которые были реализованы на языке программирования Python.

Объект исследования – задача коммивояжера.

Предмет исследования – метод ветвей и границ и метод ближайшего соседа для решения задачи коммивояжера.

Цель бакалаврской работы – анализ существующих методов решения задачи коммивояжера и реализация рассмотренных алгоритмов, выявление их достоинств и недостатков.

Для достижения цели работы необходимо решить следующие **задачи**:

1. Рассмотреть существующие методы решения задачи коммивояжера;
2. Реализовать метод ветвей и границ для решения задачи коммивояжера на языке программирования Python;
3. Реализовать метод ближайшего соседа для решения задачи коммивояжера на языке программирования Python;
4. Выявить достоинства каждого реализованного метода.

Бакалаврская работа представлена на 50 страницах, включает 27 иллюстраций, 19 таблиц, 26 формул, список используемой литературы, состоящий из 23 источников.

ABSTRACT

The title of the graduation work is «Research of various methods for solving TSP».

The object of this work is the TSP solving.

The subject of this work isthe researchmethod branch and boundand k – nearest neighbor algorithm.

The key issue of the graduation work is to find the most profitable way.

The graduation work consists of an introduction, three parts, conclusions and references. Much attention is paid to the description of the method branch and bound and k – nearest neighbor algorithm for solving travelling salesman problem.

We start with the statement of the problem, then follow through with its possible solutions, and give full coverage to method branch and bound and k – nearest neighbor algorithm.

The first part describes the travelling salesman problem, its history and algorithms for the solution of this problem, all existing solutions.

The second part considers the development of the application that solves the travelling salesman problem with the help of the k – nearest neighbor algorithm andmethod branch and bound. It contains the implementation of the algorithm, the application interface.

The third part of the work describe the test results and analysis of methods that have been implemented.

As a result, the program code was developed, described and tested to solve the traveling salesman problem with the help of the k – nearest neighbor algorithm andmethod branch and bound.

The graduation work consists of an explanatory note on 50 pages, including 27 figures, 19 tables, the list of 23 references including 6 foreign sources.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Задача о коммивояжере – относится к классу NP–полных задач дискретной оптимизации. Время работы алгоритма, решающего задачу коммивояжера, существенно зависит от размера входных данных, то есть от количества городов.

Для нее до сих пор не найдено быстрых полиномиальных алгоритмов. Для графов задача формулируется следующим образом: требуется найти гамильтонов цикл наименьшей стоимости во взвешенном полном графе. То есть каждую вершину графа нужно посетить ровно один раз.

Актуальность: Задача коммивояжера на сегодняшнее время применяется в различных сферах, таких как: маршрутизация транспортных потоков и выбор оптимальной траектории движения рабочего инструмента, а также в сферах, которые на первый взгляд не связаны с маршрутизацией: секвенирования нуклеотидных последовательностей биополимеров, эвристического определения схожести строк, построения практических алгоритмов исследования специально определённых бесконечных грамматических структур и построения эволюционных деревьев.

Объект исследования – задача коммивояжера.

Предмет исследования – метод ветвей и границ и метод ближайшего соседа для решения задачи коммивояжера.

Цель бакалаврской работы – анализ существующих методов решения задачи коммивояжера и реализация рассмотренных алгоритмов, выявление их достоинств и недостатков.

Для достижения цели работы необходимо решить следующие **задачи**:

1. Рассмотреть существующие методы решения задачи коммивояжера;
2. Реализовать метод ветвей и границ для решения задачи коммивояжера на языке программирования Python;
3. Реализовать метод ближайшего соседа для решения задачи коммивояжера на языке программирования Python;

4. Выявить достоинства каждого реализованного метода.

Обзор по главам:

1) В первой главе приводиться общая характеристика задачи коммивояжера, описание и постановки этой задачи, математическая модель и описание существующих методов решения задачи коммивояжера;

2) Во второй главе – разработка двух выбранных методов – ветвей и границ и ближайшего соседа, описание структуры программной реализации.

3) В третьей главе –проведение вычислительных экспериментов и сравнительный анализ полученных результатов.

ГЛАВА 1 ОБЗОР СУЩЕСТВУЮЩИХ МЕТОДОВ РЕШЕНИЯ ЗАДАЧИ КОММИВОЯЖЕРА

1.1 Общая характеристика задачи коммивояжера

Задача коммивояжера (путешественника или TSP – traveling salesman problem) – одна из самых известных задач комбинаторной оптимизации. Данная задача заключается в нахождении наиболее выгодного маршрута, проходящего через эти города, а затем необходимо вернуться в исходный город [12]. Задача коммивояжера в качестве линейного программирования является наиболее изученной на сегодняшний день.

В условиях задачи, указывается критерий выбора самого выгодного маршрута и соответствующая матрица расстояния, стоимости и так далее[1]. Как правило, это показывает, что маршрут должен проходить через каждый город только один раз. Поэтому данная задача имеет большую актуальность перед компаниями, которые поставляют свои товары в различные города [16].

1.1.1 Постановка задачи коммивояжера

Задача коммивояжера представляется в виде проблемы поиска в полном взвешенном графе Гамильтонова цикла минимальной стоимости [5].

Полный взвешенный граф представлен в форме

$$(F, g), \tag{1}$$

где $F = (\mathrm{V}, \mathrm{E})$;

$g: E \rightarrow N_0$.

Ограничения для частного случая проблемы (F, g)

$$M(F, g) = \{v_{i_1}, v_{i_2}, \dots, v_{i_n}, v_{i_1} | (i_1, i_2, \dots, i_n)\}, \tag{2}$$

где $(i_1, i_2, \dots, i_n)$ – некоторая перестановка чисел.

Иначе формулу 2 можно рассматривать, как множество все Гамильтоновых циклов графа F.

Стоимость для каждого цикла

$$H = v_{i_1}, v_{i_2}, \dots, v_{i_n}, v_{i_1} \in M \ F, g \tag{3}$$

определяется по формуле

$$cost \quad v_{i_1}, v_{i_2}, \dots, v_{i_n}, v_{i_1} \ , \ \mathrm{F}, g \quad = \sum_{j=1}^{n} g \quad v_{i_j}, v_{i_{j \bmod n + 1}} \ . \tag{4}$$

Стоимость каждого Гамильтонова цикла равна сумме весов всех ребер, которые входят в данный цикл.

Цель самой задачи коммивояжера – это найти минимальный путь в полном взвешенном графе[10].

Для программной реализации граф может быть представлен в виде матрицы смежности, элементы которой соответствуют весу данного ребра C_{ij}(затраченное время между данными городами), где i и j – два города, между которыми осуществляется переход. Вес или затраченное время ребра C_{ij} должно быть всегда больше 0. В дальнейшем данная матрица будет называться матрицей стоимости.

Дан полный неориентированный граф (рисунок 1).

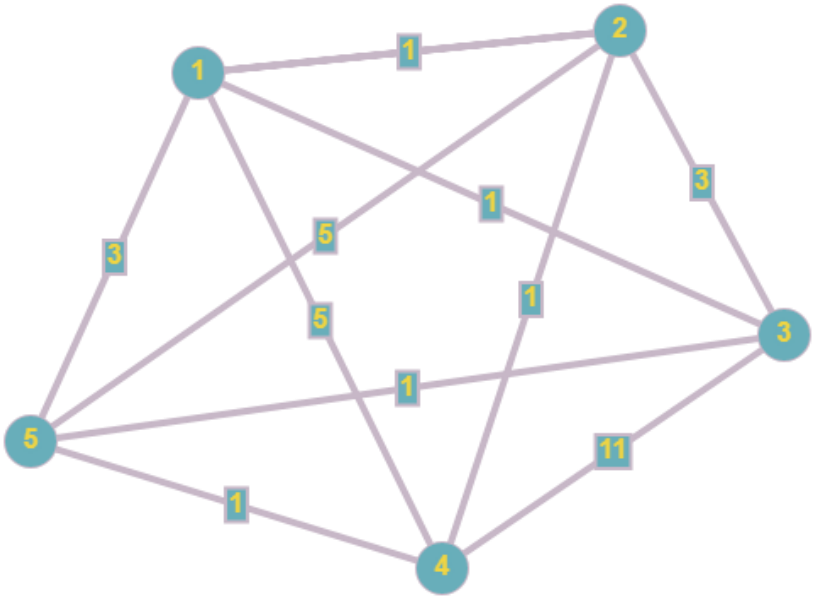

Рисунок 1 – Неориентированный граф

Приведем матрицу смежности (таблица 1), описывающую граф, изображенный на рисунке 1.

Таблица 1– Матрица смежности

	1	2	3	4	5
1	0	1	1	5	3
2	1	0	3	1	5
3	1	3	0	11	1
4	5	1	11	0	1
5	3	5	1	1	0

Всего в данном графе существуют $4!/2 = 12$ Гамильтоновых туров. Например, стоимость такого тура $H = v_1, v_2, v_3, v_4, v_5, v_1$ будет составлять $cost\ H = c\ v_1, v_2 + c\ v_2, v_3 + c\ v_3, v_4 + c\ v_4, v_5 + c\ v_5, v_1 = 1 + 3 + 11 + 1 + 3 = 19.$

В свою очередь единственным оптимальным вариантом Гамильтонова тура, является $H_{opt} = v_1, v_2, v_4, v_5, v_3, v_1,$ со стоимостью $cost\ H_{opt} = c\ v_1, v_2 + c\ v_2, v_4 + c\ v_4, v_5 + c\ v_5, v_3 + c\ v_3, v_1 = 1 + 1 + 1 + 1 + 1 = 5.$

1.1.2 Математическая модель задачи коммивояжера

Задача TSP представляет собой задачу, состоящую из целых чисел. Пусть $x_{ij} = 1$, когда торговец переходит из города i в город j. И если $x_{ij} = 0$, следовательно перехода между городами нет.

Введем город $n + 1$, расположенный в том же городе, где начинает свое путешествие торговец. Теперь из первого города можно только выйти, а в город $n + 1$ можно только зайти.

Дополнительное целочисленное значение равно количеству способов доступа к городу $u_1 = 1, u_n + 1 = n$ [13].

Чтобы избежать замкнутых путей, торговец должен покинуть первый город и вернуться в $n+1$. Определим дополнительные ограничения, которые связывают переменные x_{ij} и $u_i\ i = \{1..n\}$ – данный массив представляет собой количество всех городов, которые должен обойти торговец. Матрица C_{ij}состоит из затраченного времени между городами, где $1 \leq i, j \leq n$. Задача называется симметричной, если $C_{ij} = C_{ji}$ для всех i и j, то есть стоимость проезда или вес ребра на графе между двумя города не зависит от направления движения.

Математическая постановка задачи коммивояжера формулируется следующим образом

$$\sum_{i=1}^{n}\sum_{j=1}^{n} x_{ij}c_{ij} \rightarrow \min_{x\in\delta_\beta}, \tag{5}$$

где δ_β – это множество допустимых альтернатив, и оно представляется следующей системой ограничений:

$$\sum_{j=1}^{k} x_{ij} = 1,\ \forall i \in 1,2,\dots,n\ ; \tag{6}$$

$$\sum_{i=1}^{n} x_{ij} = 1,\ \forall j \in 1,2,\dots,n\ ; \tag{7}$$

$$u_i - u_j + nx_{ij} \leq n-1\ \ \forall i,j \in 2..n\ i \neq j\ ; \tag{8}$$

$$x_{ij} \in 0,1\ ,\ \forall i,j \in 1,2,\dots,n\ ; \tag{9}$$

$$u_i \in R^1, (\forall i \in 2,\dots,n\ ). \tag{10}$$

Формула 6 и 7 – это ограничение, которое выполняет условие: искомый путь проходит через каждую вершину графа ровно по одному разу.

Формула 8 – искомый путь должен представлять из себя единый цикл, то есть не должен распадаться на отдельные циклы.

Формула 9 – данное ограничение гарантирует, что x_{ij} принимает только булевые значения, то есть 0 или 1.

Формула 10 – ограничение, которое гарантирует, что u_i должны принимать вещественные значения [8].

1.1.3 Доказательство NP-сложности задачи коммивояжера

Теперь мы докажем, что задача коммивояжера является NP-сложной, учитывая, что задача гамильтонова цикла является NP-сложной. Эта проблема находится в списке NP-трудных задач Гэри и Джонсона и определяется следующим образом:

Задача о гамильтоновом цикле.

Задан граф $H(V, E)$. Содержит ли этот граф маршрут $T \subset E$, пройдя по которому мы сможем посетить каждую вершину $v \in V$ ровно один раз?

Вариант решения задачи коммивояжера определяется следующим образом:

Задан полный граф $G(V, E)$ длины $w_{ij} \geq 0$ для каждого ребра $(i, j) \in E$ и значение $M \in R$. Содержит ли граф маршрут длиной L такой, что $L \leq M$?

Предположим, у нас есть решение среди множества вариантов. Это решение состоит из множества рёбер T, которые составляют маршрут. Этот набор может храниться в полиномиальном пространстве. Чтобы проверить данное решение, мы должны удостовериться, что каждая вершина посещается ровно один раз и что маршрут не содержит дополнительных маршрутов. Далее мы должны просуммировать w_{ij} для всех $(i, j) \in T$ и сравнить результат с M. Это можно сделать за полиномиальное время, поэтому вариант решения принадлежит NP.

Далее докажем, что вариант решения является NP-полным путем редукции из задачи о гамильтоновом цикле. Пусть $H(V, E)$ – экземпляр гамильтонова цикла. Мы построим конкретный экземпляр задачи коммивояжера с полным графом $G\ (V, E')$, где E′ содержит неупорядоченные пары (i, j) для всех $i, j \in V$ и длины

$$w_{ij} = \begin{matrix} 0, (i,j) \in E \\ 1, \text{в другом случае} \end{matrix}. \quad (11)$$

Далее мы выбираем $M = 0$. Это сокращение можно сделать за полиномиальное время. Сейчас мы покажем, что этот экземпляр задачи коммивояжера является истинным тогда и только тогда, когда гамильтонов цикл является истинным. Предположим, что существует тур в G длины не более $M = 0$. Пусть $T \subset E'$ будет множеством из всех рёбер, которые находятся в маршруте. Из того, что все длины неотрицательны, следует, что $w_{ij} = 0$ для всех ребер $i,j \in T$. Это означает, что $T \subset E$. Ребра в T теперь образуют Гамильтонов цикл в $H(V,E)$. Теперь предположим, что в H существует Гамильтонов цикл. Обозначим T как множество ребер в цикле. Тогда $w_{ij} = 0$ для всех $(i,j) \in T$. Тогда ребра в T формируют маршрут длиной 0. Таким образом, вариант решения является NP-полным и, следовательно, задача коммивояжера является NP-сложной [4].

1.2 Методы решения задачи коммивояжера

Существует множество различных методов решения задачи коммивояжера.

1. Точные методы
    - Метод полного перебора;
    - Метод ветвей и границ;
2. Эвристические методы
    - Метод ближайшего соседа;
    - Алгоритм Метрополиса (имитация отжига);
    - Метод локальных улучшений;
    - Метод Монте – Карло (метод случайного перебора);
    - Использование генетических алгоритмов;
    - Метод муравьиных колоний;

1.2.1 Точные методы

Точные алгоритмы производят полный перебор всех вариантов. Иногда они позволяют найти решение быстро, но в основном поиск происходит по всем $n!$маршрутам, где n – количество всех городов [2]. Задача коммивояжера является трансвычислительной задачей – это означает, что при наличии более 66 пунктов в маршруте обхода, она методом перебора вариантов может решаться с помощью даже современной вычислительной техники очень большое время (около нескольких миллиардов лет) [6].

1. Алгоритм полного перебора относиться к классу методов поиска решения исчерпыванием всевозможных вариантов [3]. Сложность полного перебора напрямую зависит от количества всевозможных решений и составляет

$$O(n!). \quad (12)$$

В нашем случае, в задаче коммивояжера, сложность перебора зависит от количества городов для торговли. Минус данного алгоритма – это время, так как поиск наилучшего варианта растет экспоненциально. Алгоритм полного перебора отлично работает на небольшом количестве городов.

2. Метод ветвей и границ. С помощью такого метода можно найти точное решение задачи коммивояжера, вычисляя длины всех возможных маршрутов и выбирая маршрут с наименьшей длиной. Оценка сложности метода ветвей и границ представлена следующей формулой

$$O(n * log_2 n). \quad (13)$$

Алгоритм ветвей и границ ориентирован в большей степени на оптимизацию. В задаче уже определена числовая функция стоимости для каждой из вершин, появляющихся в дереве поиска. Цель алгоритма ветвей и

границ – это найти нужную конфигурацию, на которой функция стоимости достигает своего максимального или минимального значения [4].

Для начала рассмотрим ветвление на примере ассиметричной задачи коммивояжера, которая состоит из пяти городов (рисунок 2).

Рисунок 2 – Ассиметричный граф

В данном графе используются направления, например, из города iв город j, не обязательно такая же, как стоимость проезда между городом j в городi. Представим множество всех возможных решений в виде дерева. Корень дерева будет соответствовать множеству всех возможных путей, то есть вершина представляет множество всех 4! возможных путей в нашей задачи, состоящей из пяти городов. В общем случае для любой асимметричной задачи с N городами корень будет представлять полное множество R всех $(N-1)!$ возможных путей.

Цель алгоритма для данной задачи – это разделить весь путь на два: первый, где вероятней всего содержится оптимальный путь, второй не содержит.

Выбираем ребро (i,j) – оно скорее всего содержит оптимальный путь и разделяем R на два множества – $\{i,j\}$ и $\{\overline{i,j}\}$. В первое множество входят пути из R, содержащие ребра (i,j), а во втором не содержащие.

Составим матрицу стоимости C для нашего графа (таблица 2).

Таблица 2 – Матрица стоимости C для ассиметричного графа

	1	2	3	4	5
1	∞	25	40	31	27
2	5	∞	17	30	25
3	19	15	∞	6	1
4	9	50	24	∞	6
5	22	8	7	10	∞

Произведем ветвление на примере ребра $i, j = (3, 5)$, так как оно имеет наименьшую стоимость в матрице. Каждый путь из R содержится только в одном множестве уровня 1. В множестве $\{\overline{3, 5}\}$ не содержится оптимальный путь, как и говорилось ранее. Далее необходимо разделить второе множество $\{3, 5\}$ на следующую наименьшую стоимость в матрице – это 5 соответствующее ребру $(2, 1)$. Поэтому разделяем множества $3, 5$ на путь включающий множество $\{2, 1\}$ и не включающий – $\{\overline{2, 1}\}$.Путь от корня к любой вершине дерева выделяет определенные ребра, которые должны или не должны быть включены в множество, представленное вершиной дерева. Левая вершина уровня 2 (рисунок 3) представляет множество пути, которое содержит ребро $(3, 5)$, но не содержит ребро $(2, 1)$.

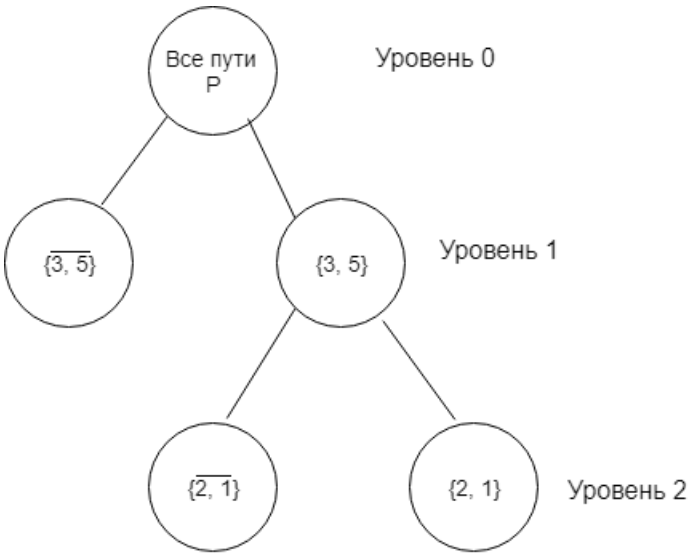

Рисунок 3 – Построение дерева

Разберем, что подразумевается под вычислением границ. С каждой вершиной дерева связывается нижняя граница стоимости любого пути из множеств, представленной вершины. В методе ветвей и границ необходимо уделить особое внимание получению как можно более точных границ. Основной шаг для вычисления границ называется приведение, он состоит из двух этапов.

1) Каждый полный путь содержит только один элемент из каждой строки и столбца, где один элемент состоит из одного ребра и соответствующей ему стоимости. Но обратное утверждение не всегда верное. Множество, которое содержит элемент, находящийся один на каждой строке и столбце, не обязательно представляет собой полный путь. Например, в матрице стоимости, представленной в таблице 2, видно, что множество $\{ 1,2 \ , \ 2,3 \ , \ 3,5 \ , \ 4,1 \ , \ 5,4 \ \}$ удовлетворяют условию одного элемента в столбце и строке, но не образует полный путь.

2) Если вычесть определенную константу k из каждого элемента строки или столбца матрицы стоимости, тогда любой путь, составленный по новой матрице C' будет меньше ровно на k чем путь, составленный по оригинальной матрице C. Такой этап называется приведение строки или столбца. Пусть opt – оптимальный путь матрицы C. Стоимость пути opt

$$z\ opt\ = \ _{(i,j)\in opt}\, C_{ij}. \quad (14)$$

Если C' получается из C приведение строки или столбца, тогда opt должен остаться оптимальным путем при C′

$$z\ opt\ = k + z'(opt), \quad (15)$$

где $z'(opt)$ – стоимость пути opt при C'.

Приведение всей матрице стоимости C – это последовательное вычитание из элементов строки константы k, а затем аналогичные действия производиться для элементов столбцов. Если для строки или столбца $h_i = 0$, то данный строка или столбец считаются приведенными, поэтому осуществляется переход к следующему строке или столбцу для последующего приведения. Осуществим приведение матрицы, представленную в таблице 3, находя минимальную стоимость в каждой строке, а затем в каждом столбце матрицы и вычитая найденную константу из каждого элемента строки или столбца (таблица 3).

Таблица 3 – Приведенная матрица стоимости C'

	1	2	3	4	5	
1	∞	0	15	3	2	$h_1 = 25$
2	0	∞	12	22	20	$h_2 = 5$
3	18	14	∞	2	0	$h_3 = 1$
4	3	44	18	∞	0	$h_4 = 6$
5	15	1	0	3	∞	$h_5 = 7$
	$h_6 = 0$	$h_7 = 0$	$h_8 = 0$	$h_9 = 3$	$h_{10} = 0$	

Значения h_i даны в конце каждой строки и столбца. Сумма всех k_i составляет 47 единиц. Следовательно, нижняя граница стоимости любого пути из R также будет равна 47

$$z\ opt\ = k + z'\ opt\ \geq k = 47. \quad (16)$$

Так как $z\ opt$ всегда больше 0 для любого пути в приведенной матрице C.

Рассмотрим нижние границы для вершин уровня 1, а именно для множеств $\{3,5\}$ и $\{\overline{3,5}\}$. Вычисление будет производиться уже над приведенной матрицей стоимости (таблица 3), не забывая, что 47 единиц, должны быть прибавлены к стоимости оптимального пути optматрицы C'.

Исходя из определения описанного выше, ребро $(3,5)$ содержится в каждом пути множества $\{3,5\}$, что препятствует выбору ребра $(5,3)$, так как ребра $(3,5)$ и $(5,3)$ образуют между собой цикл, что противоречит условию задачи коммивояжера.

В следствии чего исключается ребро $(5,3)$ из вычислений, и полагается $C_{53} = \infty$ (таблица 4).

Таблица 4 – Матрица стоимости C, без строки 3 и столбца 5.

	1	2	3	4
1	∞	0	15	3
2	0	∞	12	22
4	3	44	18	∞
5	15	1	∞	3

Далее осуществляется приведение матрицы, вычислим для каждой строки и столбца константу k (таблица 5).

Таблица 5 – Приведенная матрица стоимости C'

	1	2	3	4	
1	∞	0	3	1	0
2	0	∞	0	20	0
4	0	41	3	∞	3
5	14	0	∞	0	1
	0	0	12	2	

В данной приведенной таблице $k = 12 + 2 + 3 + 1 = 18$. Теперь нижняя граница любого пути множества $\{3, 5\}$ будет равно $47 + 18 = 65$. Нижняя граница для множества $\{\overline{3, 5}\}$ получается немного другим способом, ребро $(3,5)$ не может находиться в множестве $\{\overline{3, 5}\}$, следовательно полагаем $C_{35} = \infty$ в матрице стоимости C'. Самое дешевое ребро из города 3, исключая при этом старое значение $(3,5)$, имеет стоимость 2, а самое дешевое ребро к городу 5 равно 0. Следовательно, нижняя граница любого пути из множества $\{\overline{3, 5}\}$ получается равной $47 + 2 + 0 = 49$. Все вычисленные ранее границы указаны в дереве (рисунок 4).

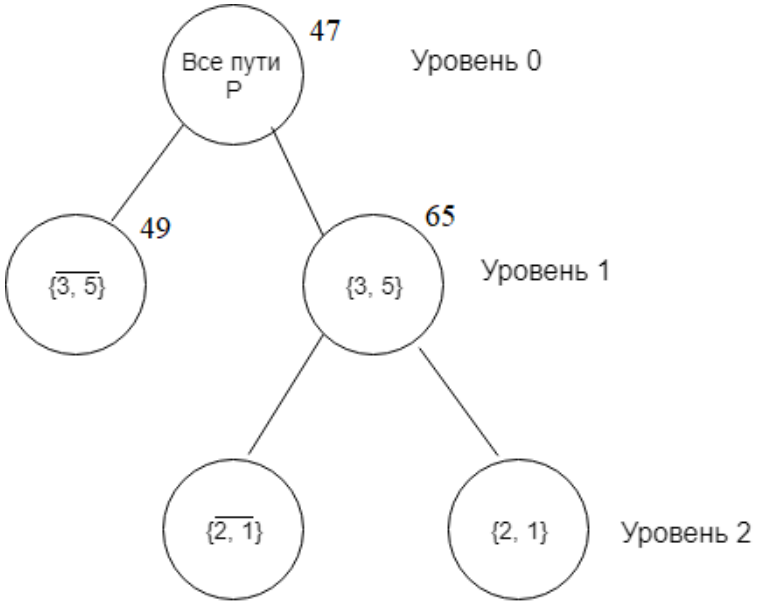

Рисунок 4 – Дерево алгоритма со всеми известными границами

На данный момент удалось сократить размер матрицы стоимости рассматриваемой вершины $\{3, 5\}$. Если найдется такой путь из множества $\{\overline{3, 5}\}$, который будет ≤ 65, тогда проводится дальнейшее ветвление и вычисление границ $\{3, 5\}$ не требуется. В данном случае такая вершина называется отработанной. Далее нужно производить ветвление над вершиной

$\{\overline{3,5}\}$, также как и ранее над $3,5$ в надежде найти путь со стоимостью в пределах $49 \leq C \leq 65$.

Представим блок-схему выполненных на данный момент времени действий (рисунок 5). Буква X обозначает текущую вершину дерева, а $w(X)$ нижнюю границу. Вершины, следующие за X обозначены Y и$\overline{Y}$, такие вершины выбираются по ребру (k, l). Обозначение стоимости самого дешевого пути, известного на данный момент – z_0.

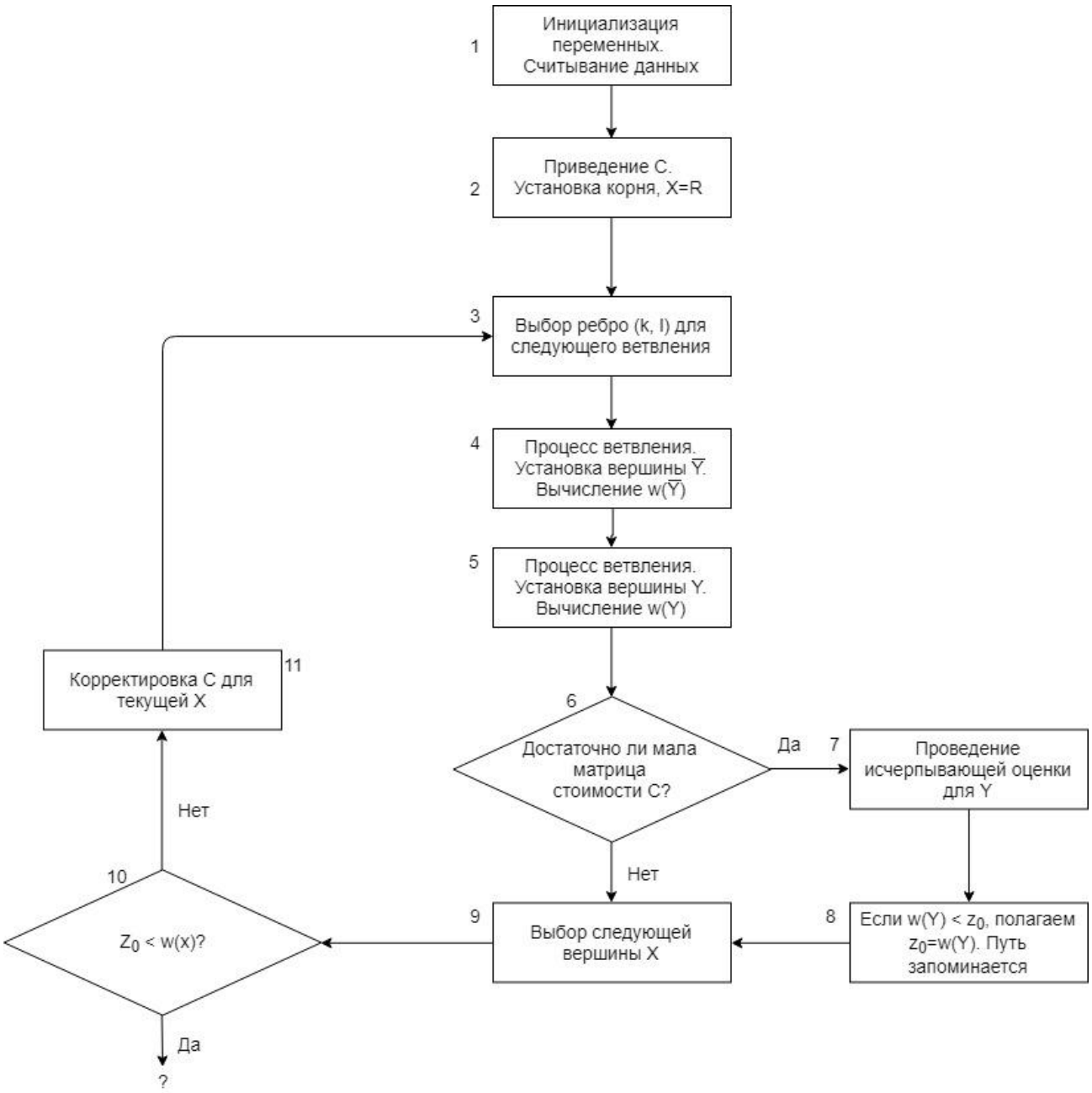

Рисунок 5 – Блок-схема алгоритма ветвей и границ

Рассмотрим блок-схему, изображенную на рисунке 5. В первом блоке показано, что нужно инициализировать переменные. Во втором блоке схемы производиться приведение матрицы C и установка корня. В 3-ем блоке – выбор последующего ребра ветвления. Подробно рассмотрим, как выбрать правильно ребро.

1. Пусть P – множество ребер (i, j), таких, что $c_{ij} = 0$ в текущей матрице стоимости C.

2. Задается D_{ij}, оно будет равно наименьшей стоимости в строке iисключая c_{ij}, и суммируется наименьшая стоимость в столбце j, исключая c_{ij}. Вычисляется D_{ij} для всех $(i, j) \in P$.

3. Далее необходимо выбрать следующее ребро ветвления (k, l) из условия $D^*_{kl} = \max_{(i,j)\in S} D_{ij}$.

Перейдем обратно к рассмотренной ранее задаче про пять городов (рисунок 2). В приведенной матрице C' первое значение X – это $w(X) = 47$. Находим первое ребро ветвления, применяя подалгоритм, который был описан выше.

1) $P = \{(1,2),(2,1),(3,5),(4,5),(5,3),(5,4)\}$.

2) $D_{12} = 2 + 1 = 3$;

$D_{21} = 12 + 3 = 15$;

$D_{35} = 2 + 0 = 2$;

$D_{45} = 3 + 0 = 3$;

$D_{53} = 0 + 12 = 12$;

$D_{54} = 0 + 2 = 2$.

3) Выбрав ребро $(k, l) = (2, 1)$, потому что данное ребро максимальное из всех остальных ребер.

В блоке 4 определяется вершина $\overline{Y}$, следующая за X. Так, в рассматриваемом примере,$X = R$ и$\overline{Y} = \{\overline{2,1}\}$ – множество всех путей, которые не содержат ребро $(2, 1)$. После вычисляется нижняя граница $w(\overline{Y})$, аналогично, как было описано выше в блоке 3

$$w\ \overline{Y}\ = w\ X\ + D_{kl}, \tag{17}$$

$$w\ \overline{\{2,1\}}\ = 47 + 15 = 62. \tag{18}$$

Рассмотрим пятый блок (рисунок 5), где вершине Y, соотвествует подмножество путей из X и содержится ребро, выбранное в блоке 3. В нашем случае – это $\{2,1\}$. Для вычисления $w(Y)$ необходимо осторожно следовать подалгоритму.

1. Из матрицы стоимости C исключаем соответствующую строку k и столбец l.

2. Введем две переменные p и q – это будут начальный и конечный город в выбранном пути. Вероятней всего, что $p = k$ или $q = l$. Также зададим $C_{pq} = \infty$, чтобы избежать замкнутого цикла, который не входит в построение любого пути.

3. Далее необходимо привести матрицу C, так же как делали это ранее.

4. Вычислить $w(Y)$ по формуле $w\ Y\ = w\ X\ + k$, где k – это константа, используемая в приведение матрицы C.

Для примера используем таблицу 3. Из данной таблицы вычеркнем вторую строку и первый столбец (таблица 6).

Таблица 6 – Матрица стоимости C, без строки 2 и столбца 1.

	2	3	4	5
1	∞	15	3	2
3	14	∞	2	0
4	44	18	∞	0
5	1	0	0	∞

Для матрицы, представленной в таблице 6, выполним операцию приведения, полученный результат представлен в таблице 7.

Таблица 7 – Приведенная матрица стоимости C'

	2	3	4	5	
1	∞	13	1	0	2
3	13	∞	2	0	0
4	43	18	∞	0	0
5	0	0	0	∞	0
	1	0	0	0	

Получаем матрицу, представленной в таблице, где k = 3 и $w(2,1) = w(X) + k = 47 + 3 = 50$. Изобразим дерево по текущему состоянию (рисунок 6).

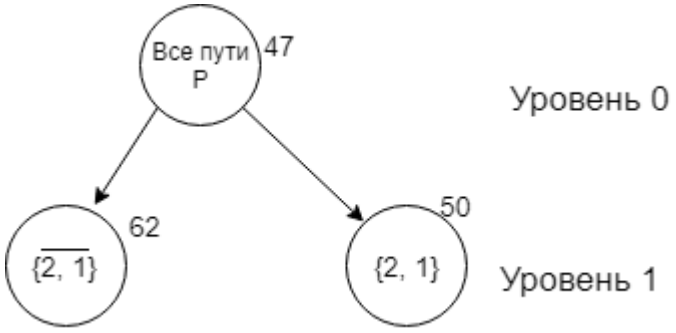

Рисунок 6 – Дерево, построенное до 1 уровня

В блоке 6 идет проверка на то, что в каждом из приведенных в итоге множеств можно рассмотреть и провести оценку без дальнейшего ветвления. Каждое ребро, которое содержится во всех путях Y, сокращает размер матрицы стоимости Cна одну строку и один столбец.

Переход на блок 7 осуществляется только тогда, когда матрица стоимости приведена до конца, то есть проверка в блоке 6 прошла. Данный блок ищет самый дешевый путь в Y и его вес обозначается как $w(Y)$.

В блоке 8 проверяется, лучший ли данный путь, чем предыдущий (текущий) из всех известных путей. Если нет, тогда данный путь отбрасывается, иначе становиться лучшим путем, полагая, что $z_0 = w(Y)$.

Выбор следующей вершины для последующих ветвлений осуществляется в блоке 9. Выбирается вершина, из которой не выходят на

данный момент ветви, и она имеет самую меньшую по стоимости нижнюю границу.

1. Ищем множество P конечных вершин текущего дерева поиска.

2. Вершина X выбирается по формуле

$$w\ X\ = \min_{v \in P} w(v). \quad (19)$$

Рассмотрим данные шаги на примере задачи про пять городов.

1) $P\ = \{\ \overline{2,1}\ ,\ 2,1\ \}$.

2) $w\ \ \overline{2,1}\ \ = 62;$

$w\ \ 2,1\ \ = 50.$

3) Следовательно $X\ =\ \ 2,1\ .$

В блоке 10 спрашивается должен ли остановиться алгоритм, или продолжать выполнять его. Если текущая граница лучшего пути удовлетворяет условию $z_0 \leq w(X)$, тогда ни одна из последующих границ, не может содержать лучший путь. Но благодаря выбору X в блоке 9, выбранный лучший путь не может содержаться ни в какой другой из неоцененных конечных вершин. Если $w\ X\ < z_0$, то работа алгоритма продолжается.

Откорректированная матрица стоимости C получается в блоке 11.

1) Если множество X равно множеству Y, то текущая матрица C – это то, что нужно, и переходим в блок 3.

2) Текущая матрица C теперь считается исходной матрицей стоимостей.

3) Множество всех пар (i, j), которые должны быть ребрами в X, входят в P.

4) Вычисляем $g\ =\ \ _{(i,j)\in R}\ C_{ij}$.

5) Для каждого ребра $(i, j) \in P$ вычеркивается строка i и столбецj в текущей матрице C.

6) Матрицу C приводим и задаем kравным сумме констант приведения.

7) На конечном шаге вычисляем $w\ X\ = g + k$.

Продолжим рассматривать пример, находясь в третьем блоке после возвращения из блока одиннадцать. $X = \{1, 2\}$ и $w(X) = 50$, также наше дерево содержит два уровня: нулевой и первый (рисунок 6). Далее используется алгоритм, описанный выше в блоке 3.

1) $P = (\{1, 5\}, \{3, 5\}, \{4, 5\}, \{5, 2\}, \{5, 3\}, \{5, 4\})$;

$D_{15} = 1 + 0 = 1$;

$D_{35} = 2 + 0 = 2$;

$D_{45} = 18 + 0 = 18$;

$D_{52} = 13 + 0 = 13$;

$D_{53} = 13 + 0 = 13$;

$D_{54} = 0 + 1 = 1$.

3) $D_{kl} = D_{45} = 18$, если таких ребер несколько, то выбираем любое. Таким образом $(k, l) = (4, 5)$.

Переходим к блоку 4, устанавливая $\overline{Y} = (\{\overline{4, 5}\})$ и $w\ \ 2,1\ \ + D_{45} = 50 + 18 = 68$. Рассмотрим вершину $(4,5)$, убираем из матрицы C строку с номером 4 и столбец под номером 5, так как выбранное ребро не пересекается с $(2, 1)$, тогда $p = 4$ и $q = 5$. Также ставим в ячейку $C_{54} = \infty$ и строим матрицу стоимости C (таблица 8).

Таблица 8 – Матрица стоимости C, без строки 4 и столбца 5.

	2	3	4
1	∞	13	1
3	13	∞	2
5	0	0	∞

Далее осуществляется приведение матрицы, которая представлена в таблице 8, для этого вычисляем $k = 2 + 1 = 3$ (таблица 9).

Таблица 9 – Приведенная матрица стоимости C'

	2	3	4	
1	∞	12	0	1
		25		

3	11	∞	0	2
5	0	0	∞	0
	0	0	0	

После переходим к блоку 6, определяем является ли текущая матрица C конечной, то есть 2×2. Матрица в таблице 9 является 3×3, следовательно переходим к блоку 9.

1)$P = (\{\overline{2,1}\}, \{\overline{4,5}\}, \{4,5\})$.

2) $w\ \ \overline{2,1}\ \ = 62;$

$w\ \ \overline{4,5}\ \ = 68;$

$w\ \ 4,5\ \ = 53.$

Из второго шага следует, что $X = \{4,5\}$. Изобразим текущую ситуацию в виде дерева алгоритма (рисунок 7).

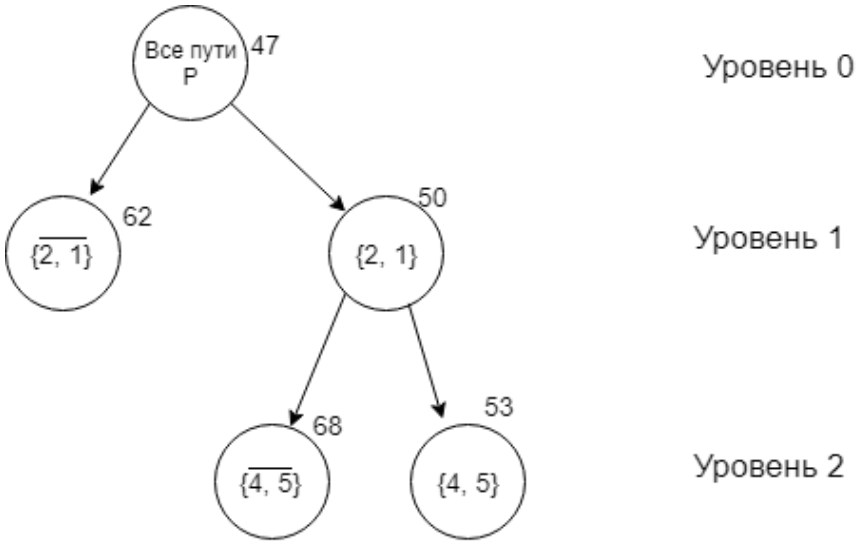

Рисунок 7 – Дерево, построенное до 2 уровня

Пропускаем в блок-схеме блок 10, так как $z_0 = \infty$. Дальше возвращаемся в блок 3, используя полученные данные на предыдущем шаге и матрицуC, представленную в таблице 8.

1) $P = (\ 1,4\ ,\ 3,4\ ,\ 5,2\ ,\{5,3\})$.

2) $D_{14} = 12 + 0 = 12;$

$D_{34} = 11 + 0 = 11;$

$D_{52} = 11 + 0 = 11;$

$D_{53} = 12 + 0 = 12.$

3) $D_{kl} = D_{14} = D_{53} = 12$. Получилось два ребра одинаковые по весу, поэтому выбираем любое, например, ребро $(5, 3)$.

Теперь $\overline{Y} = \{\overline{5, 3}\}$ и $w\ \overline{Y}\ = w\ \ 4, 5\ \ + D_{14} = 53 + 12 = 65$. Так как $Y = \{5, 3\}$, следовательно, необходимо вычеркивать пятую строку и третий столбец. $C_{35} = \infty$ для исключения образования цикла.

Произведем приведение матрицы C (таблица 10). Из таблицы 9, видно, что $k = 11$, а $w(Y) = 53 + k = 53 + 11 = 64$.

Таблица 10 – Матрица стоимости C, без строки 5 и столбца 3.

	2	4	
1	∞	0	0
3	11	0	0
	11	0	

Сумма констант приведения матрицы 10 равняется 11. Нижняя граница $5{,}3\ = 54 + 11 = 65 < 66$, поскольку нижняя граница подмножества меньше, чем подмножество $\{\overline{5, 3}\}$, тогда ребро $5{,}3$ включается в маршрут с новой границе 65.

Далее берем ребро $(1, 4)$, так как D_{14} самое наименьше. $\overline{Y} = \{\overline{1, 4}\}$и $w\ \overline{Y}\ = w\ \ 4, 5\ \ + D_{14} = 53 + 12 = 65$. Так как $Y = \{1, 4\}$, следовательно необходимо вычеркивать первую строку и четвертый столбец. Ребро $(1, 4)$ образует путь вместе с ребрами $(2, 1)$ и $(4, 5)$. Так как $p = 2 и q = 5$, поэтому $C_{52} = \infty$, приведем матрицу C(таблица 11). Из таблицы 9, видно, что $k = 11$, а $w(Y) = 53 + k = 53 + 11 = 64$.

Таблица 11 – Матрица стоимости C, без строки 1 и столбца 4.

	2	3

3	0	∞	11
5	∞	0	0
	0	0	

Переходим в блок 6 с матрицей C(таблица 10) размера 2×2, данный блок выдает путь со следующим порядком: 3 2 1 4 5 3 и стоимостью $z = 64$. В блоке 8 с учетом того, что $z_0 = 64$ получаем лучший текущий путь. После не нужно рассматривать вершины следующие за $\{4, 5\}$ и $\{1, 4\}$, так как их нижние границы больше чем 64. Изобразим дерево на текущий момент решения задачи (рисунок 8).

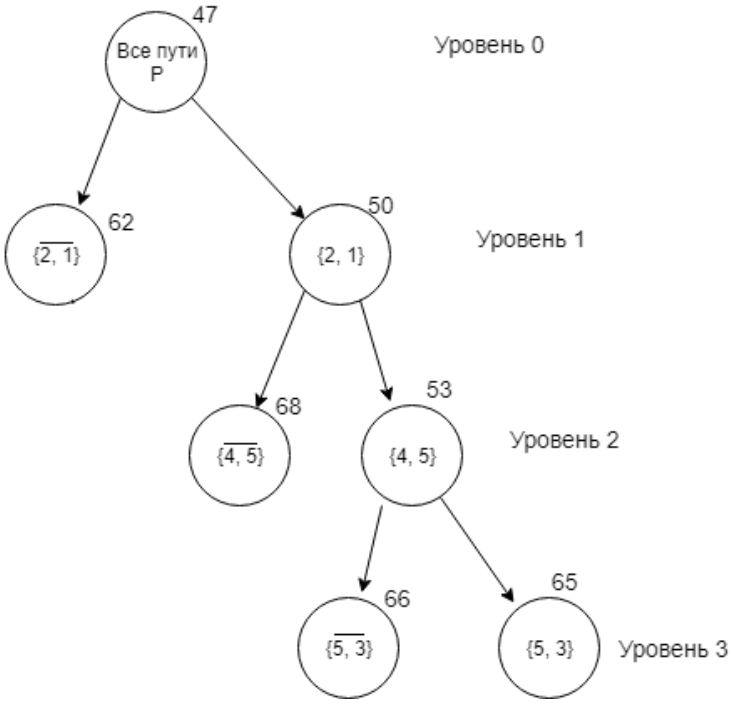

Рисунок 8 – Дерево, построенное до 3 уровня

В блоке 10 видно, что у конечной вершины $\{\overline{2, 1}\}$нижняя граница равна 62, что соответственно меньше, чем 64. Поэтому возможно в данной вершине может находится путь лучший, чем текущий. Находим $X = \{\overline{2, 1}\}$,$w(X) = 62$и матрицуC, приведенную с помощью алгоритма, описанного в блоке 11 (таблица 12).

Таблица 12 – Приведенная матрица стоимости C'

	1	2	3	4	5
1	∞	0	15	3	2
2	∞	∞	0	10	8
3	15	14	∞	2	0
4	0	44	18	∞	0
5	12	1	0	0	∞

В блоке 3 выбираем следующее ребро для разветвления $4,1$ с $D_{41} = 12$. В блоках 4 и 5 вычисляется $w\ \overline{4,1}\ = 62 + 12 = 74$ и $w\ \ 4,1\ \ = 62 + 3 = 65$. Из вычислений видно, что $w\ \ 4,1\ \ = 74$ больше, чем $w\ \ 1,4\ \ = 64$, следовательно, $z = 64$ остается лучшим путем. Изобразим полученное дерево на рисунке 9.

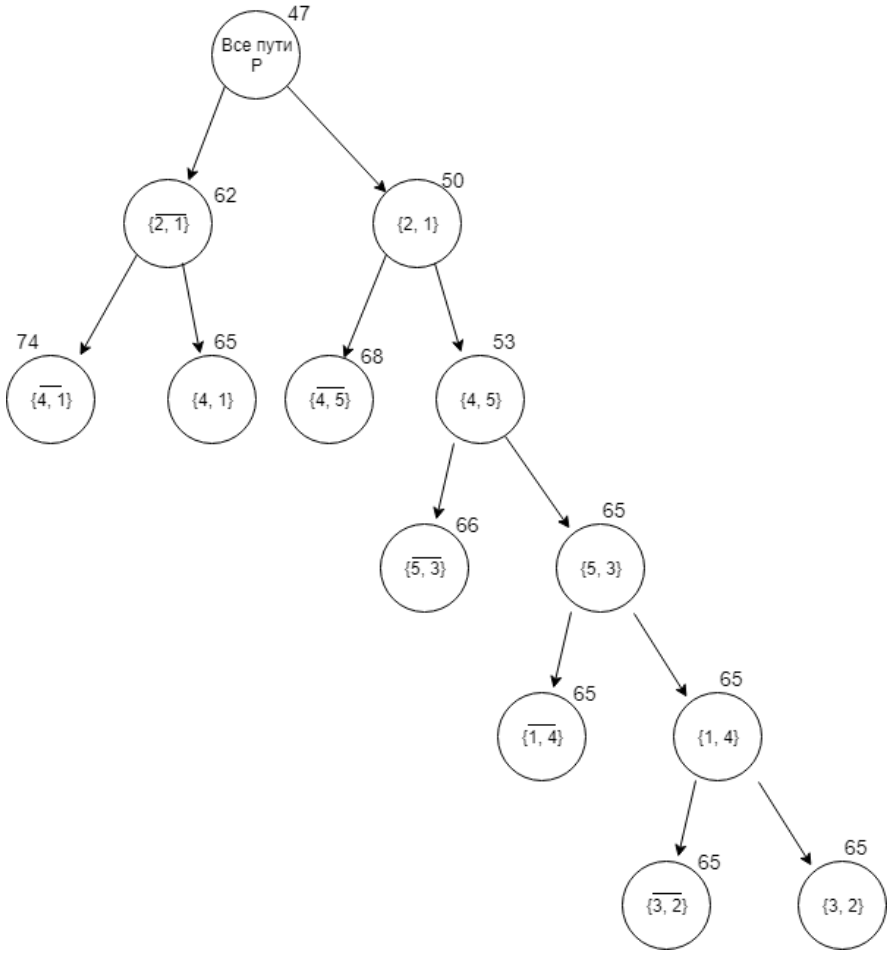

Рисунок 9 – Окончательный вариант дерева

В результате работы метода ветвей и границ в примере задачи про пять городов гамильтонов цикл образуют ребра: $2,1$, $1,4$, $4,5$, $5,3$,$\{3,2\}$ – 2 1 4 5 3, с длиной маршрута равной 64.

1.2.2 Приближенные и эвристические методы

Приближенные методы решения задачи коммивояжёра относятся к эвристическим методам и являются довольно эффективными, т.к. сокращают полный перебор маршрутов. Во многих из них находят не эффективный маршрут, а базовый маршрут, т.е. приближённое решение. В дальнейшем этот базовый маршрут улучшается. Жадный алгоритм – это алгоритм, который предполагает принятие локальных оптимальных решений на каждом этапе. Также допуская, что окончательное решение будет оптимальным [14].

1. Алгоритм ближайшего соседа. На первом шаге выбираем город (т.е. вершину графа), из которого выходит наименьшая длина дороги (ребра). На последующих шагах выбираем ближайший город (вершину), который еще не проходили. Сложность данного алгоритма выражается формулой

$$O(n). \qquad (20)$$

Составим матрицу симметричную относительно главной диагонали (таблица 13). Данная матрица была сгенерирована случайным образом.

Таблица 13– Матрица смежности

	1	2	3	4	5	6	7
1	∞	6	23	8	43	26	15
2	6	∞	21	3	40	24	5
3	23	21	∞	20	22	4	17
4	8	3	20	∞	38	22	4
5	43	40	22	38	∞	18	35
6	26	24	4	22	18	∞	19
7	15	5	17	4	35	19	∞

На рисунке 10 представлен граф для таблицы 13.

Рисунок 10 – Граф

Теперь выбираем с помощью генератора случайных чисел первый город – 4. Далее ищем самый короткий путь в другой возможный город, то есть ребро с наименьшим весом. Ребро ведущие из города 4 в город 2 с весом 3.

1. 4 – 2 = 4
2. 2 – 1 = 6
3. 1 – 7 = 15
4. 7 – 3 = 17
5. 3 – 6 = 4
6. 6 – 5 = 18
7. 5 – 4 = 38

Посчитаем общую длину всего пути = 102. Это и есть разбор метода ближайшего соседа. Блок-схема алгоритма ближайшего соседа представлена на рисунке 11.

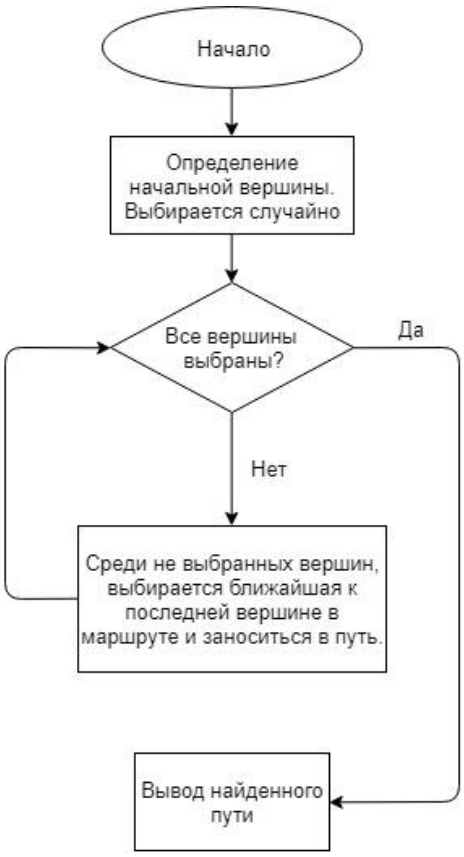

Рисунок 11 – Блок-схема метода ближайшего соседа

2. Алгоритм Метрополиса (имитация отжига) основывается на имитации физического процесса, который происходит при кристаллизации вещества из жидкого состояния в твёрдое, в том числе при отжиге металлов [21]. Оценка сложности алгоритма имитации отжига показана в формуле

$$O(n^2 * log_2 n), \quad (21)$$

гдеn –количество городов.

Основные шаги алгоритма.

1) Выбор начального решения и начальной температуры;
2) Оценка начального решения;
3) Основной шаг алгоритма;
    - Случайное изменение текущего решения;
    - Оценка измененного решения;
    - Критерий допуска;

4) Уменьшение температуры и, если температура больше некоторого порога, то переход к шагу 3.

В начале алгоритма следует выбрать начальную и минимальную температуру. Начальная температура t_{max} – это начало отжига, а минимальная t_{min} – это конец отжига. Задается произвольное начальное состояние системы S_0, тем самым определяется функция для нового состояния.

Далее следует сравнить два значения целевой функции – текущее и после принятия нового значения – $E\ S_0$ и $E\ S_i$ соответственно. Если вычисленное значение целевой функции $E\ S_i$ оказалось меньше, чем текущее, тогда сохраняем текущее в качестве целевой функции $E\ S_i < E\ S_0 \rightarrow S_0 = S_i$. Данное правило применяется для задачи на поиск глобального минимума. Если задача на поиск максимума, тогда формула будет выглядеть так: $E\ S_i > E\ S_0 \rightarrow S_0 = S_i$ [17]. Вероятность принятия текущего значения вычисляется по формуле

$$P = e^{\frac{-\delta E}{T}}, \tag{22}$$

где P – вероятность принятия значения;

$\delta E = E\ S_0 - E\ S_i$, S_0 – текущее состояние, S_i – новое принятое состояние;

T_i – текущая температура.

Если вычисленная вероятность больше определенного значения, например, 0.5, тогда сохраняем новое состояние системы $S_0 = S_i$. После вычисляется новое значение температуры T_i, она может понижаться в зависимости от закономерности рассматриваемой задачи, например линейной. Линейную закономерность можно представить в виде формулы

$$T_i = T_{i-1} * r/j, \tag{23}$$

где j – шаг итерации;

r – коэффициент, который настраивается и обычно принимает значение от 0,8 до0,99.

Если выполняется условие $T_i < T_{min}$, тогда алгоритм заканчивается, иначе переходим к выбору нового состояния. Блок-схема алгоритма отжига, представлена на рисунке 12 [22].

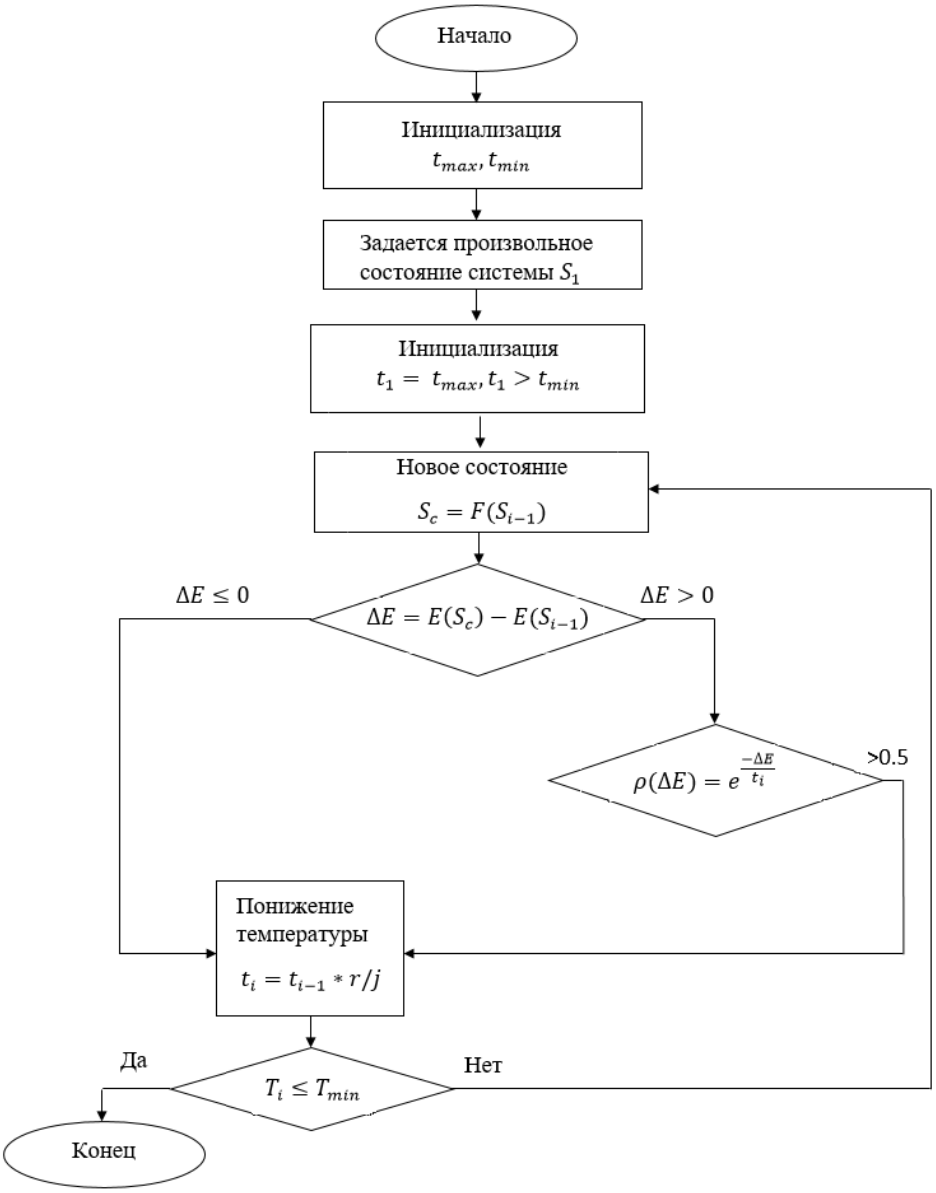

Рисунок 12 – Блок-схема алгоритма отжига

3. Генетический алгоритм – это алгоритм поиска, который можно применять для решения задачи оптимизации. Генетический алгоритм использует методы аналогичные естественному отбору в природе, такие как наследование, мутация, кроссинговер и отбор [18]. Сложность генетического алгоритма представлена в формуле

$$O(t * m * n^2), \quad (24)$$

где n – количество городов;

m – численность популяции;

t – количество итераций.

В начале работы алгоритма необходимо задать целевую функцию, функцию приспособленности для популяции, и начальную популяцию, которая создается случайным образом.

Затем определяется скрещивание – берется два гена разных особей, впоследствии они называются родителями, скрещиваться таким образом, чтобы потомки от данных родителей унаследовали черты каждого из родителей [19].

Следующим шагом происходит мутация полученных потомков и этап отбора. Отбор происходит зависимости от функции принадлежности из особей, полученных на предыдущих этапах, выживает только часть [20].

При применении генетического алгоритма для решения задачи коммивояжера, вместо функции приспособленности, для каждой особи используется длина пути объезда городов[15].

Входными данными задачи коммивояжера являются координаты городов. С помощью этих координат мы получаем матрицу расстояний между городами. Блок-схема генетического алгоритма изображена на рисунке 13.

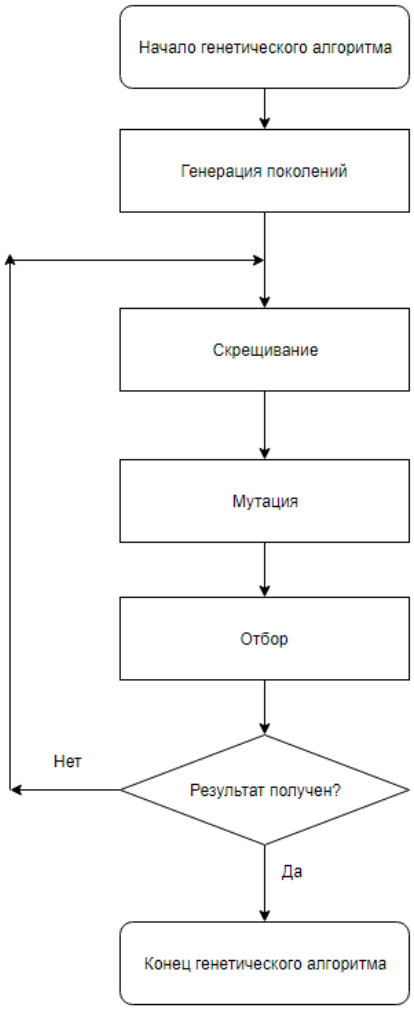

Рисунок 13 – Блок-схема генетического алгоритма

4. Алгоритм муравьиной колонии – один из эффективных полиномиальных алгоритмов для нахождения приближенных решений задачи коммивояжера [23].

Временная сложность алгоритма совпадает с генетическим алгоритмом и составляет

$$O(t * m * n^2), \quad (25)$$

где n –количество городов;

m – численность популяции;

t – количество итераций.

Блок-схема алгоритма муравьиной колонии, представлена на рисунке 14.

Рисунок 14 – Блок-схема алгоритма муравьиной колонии

В основе данного метода лежит поведение муравьев в живой природе. В поисках пищи муравей проходит случайный путь от гнезда до месторасположения еды, а после, возвращаясь, оставляет за собой след из феромона.

Этот феромон привлекает других муравьев, находящихся вблизи к следу из феромона, который, вероятней всего, привлечет других муравьев, и они пройдут по данному следу, тем самым возвращаясь в гнездо и укрепляя феромонную тропу. Если же существует два и более маршрута от еды до гнезда, то по более короткому успеют пройти больше муравьев, чем по

длинному. И тогда короткий путь станет привлекательнее для муравьев из-за большого количества феромона. Длинные же пути в итоге исчезнут из-за испарения.

Таким образом, были рассмотрены основные методы решения задачи коммивояжера. Представим сводную таблицу по основным методам (таблица 14).

Таблица 14 – Сложность основных методов решения задачи коммивояжера

Метод решения задачи коммивояжера	Сложность метода
Точный метод	$O(n!)$
Метод ветвей и границ	$O(n * log_2 n)$
Алгоритм ближайшего соседа	$O(n)$
Алгоритм Метрополиса (имитация отжига)	$O(n^2 * log_2 n)$
Генетический алгоритм	$O(t * m * n^2)$
Алгоритм муравьиной колонии	$O(t * m * n^2)$

В настоящее время существует множество модификаций методов решения задачи коммивояжера.

Для реализации методов решения задачи коммивояжера были выбраны два метода: ближайшего соседа и ветвей и границ.

ГЛАВА 2 РЕАЛИЗАЦИЯ АЛГОРИТМОВ РЕШЕНИЯ ЗАДАЧИ КОММИВОЯЖЕРА

2.1 Структура программы

Алгоритмы были реализованы вPyCharm – это интегрированная среда разработки для языка программирования Python.PyCharm была разработана компанией JetBrains.

Реализация была разработана на версии Python 3.7. Были использованы следующие библиотеки:

1. matplotlib 3.1.0
2. numpy 1.16.4

Для вычисления оптимального маршрута в задаче коммивояжера в программе реализован метод ближайшего соседа и метод ветвей и границ.

Проведем краткий обзор по важным составляющим программы:

1. Метод для создания матрицы $N \times N$;
2. Класс для решения методом ближайшего соседа;
3. Функция для создания случайного оттенка вершины, для легкого распознавания в графике;
4. Функция проверки на ошибки, то есть не корректный ввод входных данных;
5. Функция для вывода необходимых данных, таких как время работы алгоритма, сумма всего пути, начальная и конечная вершина;
6. Реализация метода ближайшего соседа для решения задачи коммивояжера;
7. Класс для хранения всех узлов дерева;
8. Класс для решения методом ветвей и границ;
9. В основной части кода реализован пользовательский интерфейс программы.

2.2 Реализация метода ближайшего соседа

Для решения задачи коммивояжера метода ближайшего соседа необходимо для начала сгенерировать матрицу стоимости всех вершин. На вход программы через пользовательский интерфейс подается одно число – количество вершин, так как матрица симметричная относительно главной диагонали.

Класс MyMatrix содержит следующие поля:

- $beforeN$– это поле, в которое записывает последнее введенное число вершин через интерфейс пользователя;
- X и Y– это два массива, в которых случайным образом генерируются точки в графе, где X соответствует оси абсцисс, а Yординат;
- $matrix$– массив для матрицы стоимости.

Матрица создается при помощи созданного метода generateMatrix в классе MyMatrix. Матрица стоимости создается и записывается в массив $matrix$. Заполнение матрицы происходит с помощью формулы Евклидового расстояния

$$d\ p,q\ = \ \overline{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2}, \qquad (26)$$

где pиq – это первая и вторая точка на графике соответственно [11].

А главная диагональ заполняется inf, так как циклы в данном графе присутствовать не могу по определению задачи коммивояжера.

На рисунке 15 представлена диаграмма класса MyMatrix.

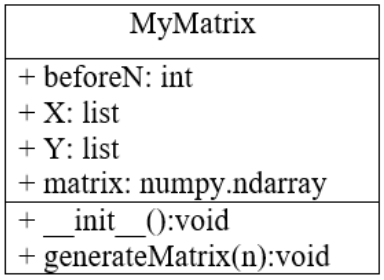

Рисунок 15 – UML-диаграмма класса MyMatrix

Далее создается класс NearestNeighbor – это класс для решения задачи коммивояжера методом ближайшего соседа. На рисунке 16 представлена диаграмма класса.

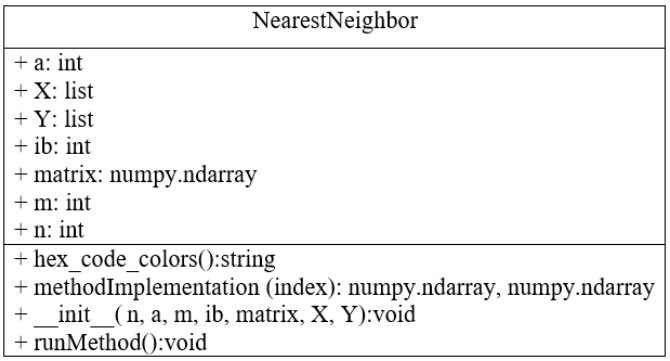

Рисунок 16 – UML-диаграмма класса NearestNeighbor

Реализация ближайшего соседа представлена в методе methodImplementation. На вход метода подается индекс вершины графа, а на выходе массив с суммой найденного пути и массив с индексами оптимального пути. Также в данном методе присутствует модификация. В главе 1 в описание метода ближайшего соседа первая вершина графа (первый город в пути) выбирается случайным образом, в моей реализации каждый город выбирается поочередно, после вычисляется сумма построенного маршрута. В итоге выводится на экран тот маршрут, который получился самым минимальным по сумме.

В методе runMethodреализован вывод графика, он выводиться на экран с помощью библиотеки matplotlib, в коде название библиотеки сокращено до plt для удобства в использовании. За генерацию случайных цветов для точек на графе отвечает метод hex_code_colors.

Фиксирование времени начинается с вызова функции с реализацией ближайшего соседа и заканчивается тогда, когда функция пройдет все nточек. Время работы алгоритма и сумма построенного маршрута выводятся на экран с помощью встроенного пользовательского интерфейса tkinter.

Для запуска реализованного метода в пользовательском интерфейсе присутствует кнопка «Nearest neighbor», она запускает метод

runNearestNeighbor, который проверяет на корректность ввода количества вершин (N), а после того, как ввод будет считать корректным запускается созданный ранее класс NearestNeighbor.

Весь описанный алгоритм программы представлен в Приложении А.

2.3 Реализация метода ветвей и границ

Для решения задачи коммивояжера метод ветвей и границ необходимо использовать созданную ранее матрицу в методе ветвей и границ. На вход программы через пользовательский интерфейс подается одно число – количество вершин.

Создается класс Node для всех узлов в дереве для метода ветвей и границ. На рисунке 17 представлена диаграмма класса Node.

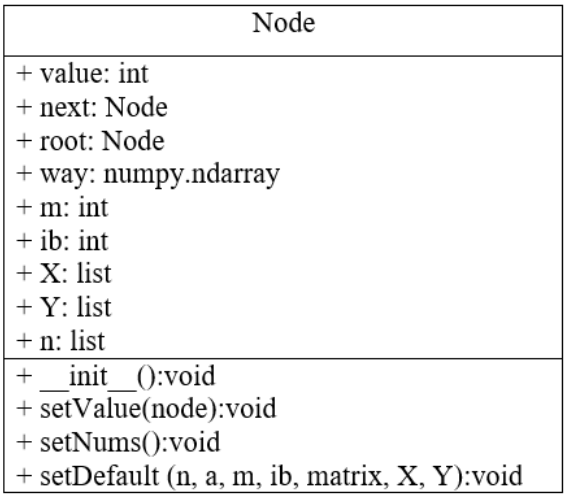

Рисунок 17 – UML-диаграмма класса Node

Данный класс состоит из конструктора, метода setValue, метода setNums и метода setDefault.

Метод setDefault задает начальное значение для всех переменных. На вход принимает следующие параметры:

- n – количество вершин на графе;

- a и m– координаты вершин задаются случайным образом от a до m;
- ib – целочисленный номер первой вершины;
- $matrix$– массив, матрица стоимости;
- Xи Y– это два массива, в которых случайным образом генерируются точки в графе, где X соответствует оси абсцисс, а Y ординат.

Метод setValue копирует значение текущего узла, строки, столбца и матрицы, принимая на вход узел.

Для правильного отображения путей нам необходима матрица со всеми номерами строк и таблиц, так как при выполнении метода ветвей и границ на каждой итерации вычеркивается найденная строка и столбец в матрице стоимости. Данный алгоритм описан в методе setNums.

Следующий класс BnP – это класс для решения задачи коммивояжера методом ветвей и границ. Все методы в данном классе статические, такие методы не привязаны к конкретному объекту. В статических методах вызовы функций происходят через объявление класса, а не через объект. Диаграмма класса BnP– реализация метода ветвей и границ (рисунок 18).

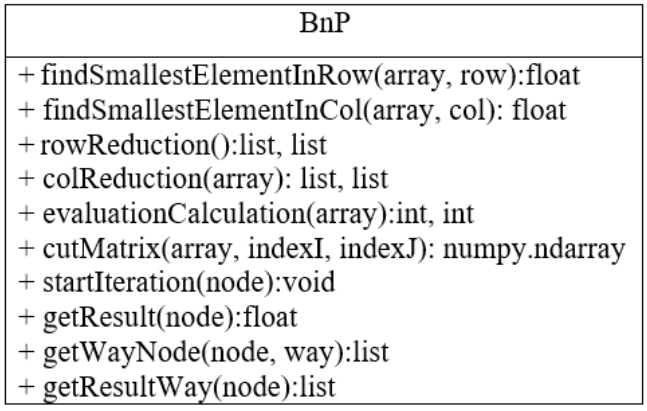

Рисунок 18 – UML-диаграмма класса BnP

Метод для нахождения наименьшего значения в строке – findSmallestElementInRow, он принимает на вход матрицу стоимости и номер строки.

Метод findSmallestElementInCol создан для нахождения наименьшего значения в столбце матрицы стоимости, аналогично методу findSmallestElementInRow, но за исключением того, что он принимает на вход вместо строки, столбец.

В теории, описанной в 1 главе, после нахождения минимума в строках и столбцах происходит приведение матрицы. Данная операция реализована в методах rowReduction и colReduction, которые находят минимум в строках и в столбцах соответственно.

Далее объявляется метод evaluationCalculation. Он служит для оценки нулевых точек. Согласно теории, до вычеркивания сток и столбцов в матрице, необходимо найти максимальную точку, для дальнейшего ветвления дерева.

В классе также присутствует метод, который служит для вычеркивания максимальной строки и столбца в матрице – cutMatrix. Данный метод принимает на вход матрицу стоимости, индекс по строкам и индекс по столбцам.

В startIteration проводиться одна итерация метода ветвей и границ, принимая на вход только узел дерева. Итерация состоит из нескольких шагов:

1. Нахождение минимума по строкам и столбцам;
2. Считывается полный путь, учитывая путь предыдущего значения;
3. Нахождение нуля с максимальной оценкой;
    3.1. Задается путь к текущему нулю, равняется бесконечности;
    3.2. Удаление текущего маршрута;
4. Все маршруты заносятся в массив путей очередной путь.

Следующий метод в классе getResult рекурсивно выдает сумму найденного маршрута. На вход принимается узел из дерева.

Также в классе реализован метод, который рекурсивно собирает все маршруты в один – getWayNode.

Последний реализованный метод в классе BnP – это getResultWay. Данный метод вычисляет оптимальный вектор оптимального пути. На вход принимается значение текущего узла, а возвращает максимальный маршрут, который был найдет во входной матрице стоимости.

Фиксирование времени начинается с вызова метода startIterationи заканчивается тогда, когда метод закончится. Время работы алгоритма и сумма построенного маршрута выводятся также на экран с помощью встроенного пользовательского интерфейса tkinter.

Для запуска реализованного метода в пользовательском интерфейсе присутствует кнопка «BnP», она запускает метод runBnP, который проверяет на корректность ввода количества вершин (N), а после того, как ввод будет считать корректным запускается созданный ранее класс NearestNeighbor.

В методе runBnP также присутствует вывод графика с помощью библиотеки matplotlib, аналогично методу runMethod из класса NearestNeighbor.

В пользовательском интерфейсе также присутствует кнопка «Creatematrix», данная кнопка служит для генерации матрицы стоимости случайным образом, который был описан выше в методе generateMatrix из класса MyMatrix.

Весь описанный алгоритм программы представлен в Приложении А.

2.4 Интерфейс приложения

Для удобного пользования был разработан графический интерфейс пользователя. С помощью встроенной библиотеки tkinter. Пример работы графического интерфейса представлен на рисунке 19.

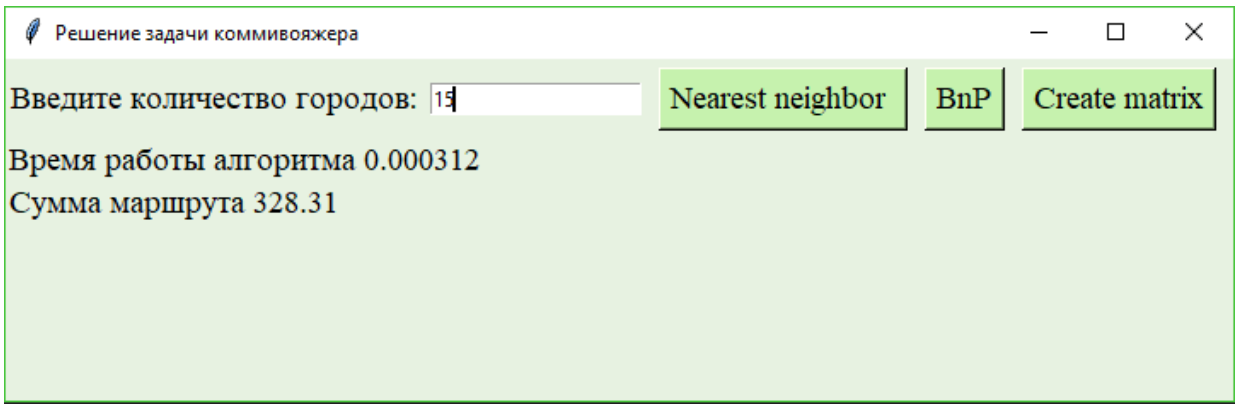

Рисунок 19 – Пример работы графического интерфейса

Также при неправильном вводе входных данных, например, ноль или один город, буквы или различные символы, программы выдает сообщение об ошибке (рисунок 20).

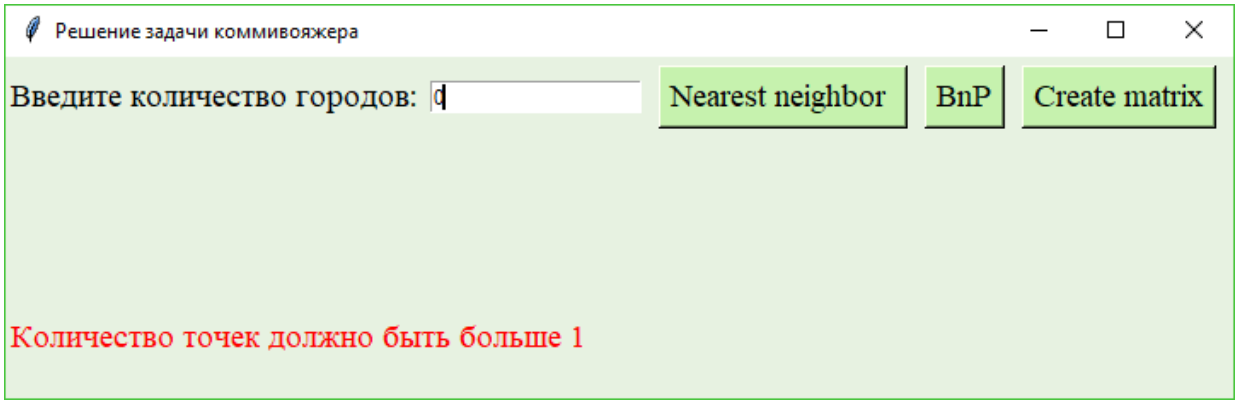

Рисунок 20 – Пример неправильно введенных данных

ГЛАВА 3 СРАВНИТЕЛЬНЫЙ АНАЛИЗ РЕАЛИЗАЦИЙ

3.1 Технические данные для сравнительного анализа

Вычислительные эксперименты проводились на персональном компьютере со следующими характеристиками (таблица 15).

Таблица 15 – Технические данные ПК

Процессор	IntelCorei5-6600 CPU 3.30 GHz
Память	8 Gb
Операционная система	Microsoft Windows 10

Результатом работы программы будет график с оптимальным путем и засеченным временем работы алгоритма. Для каждого разработанного алгоритма матрица должна содержать одинаковые значения, для точного анализа.

При разработке программы были предусмотрены методы, которые отвечают за подсчёт времени работы алгоритма в миллисекундах.

3.2 Метод ближайшего соседа

Разработанная программа позволяет задать количество вершин n, координаты вершин генерируются случайным образом, то есть структура матрицы генерируется каждый раз новым образом с весами ребер.

Во время работы метода ближайшего соседа фиксировалось время. Также программа выдает сумму всего построенного пути. Исследования проводились до $N = 1000$, все данные представлены в таблице 16.

Таблица 16 – Результаты расчета маршрута

N	Время работы (миллисекунды)	Сумма всего пути
5	0,000059	211,42
10	0,000227	232,03
15	0,000315	362,38
20	0,000680	421,46
25	0,000773	460,23
30	0,001077	514,65
35	0,001437	510,7
40	0,001748	540,16
45	0,002623	564,79
50	0,002625	669,89
100	0,010576	888,41
200	0,040370	1233,23
300	0,094620	1505,75
400	0,147421	1748,69
500	0,232296	1937,36
1000	1,109630	2716,27

На рисунке 21 представлена зависимость времени от количества N в матрице.

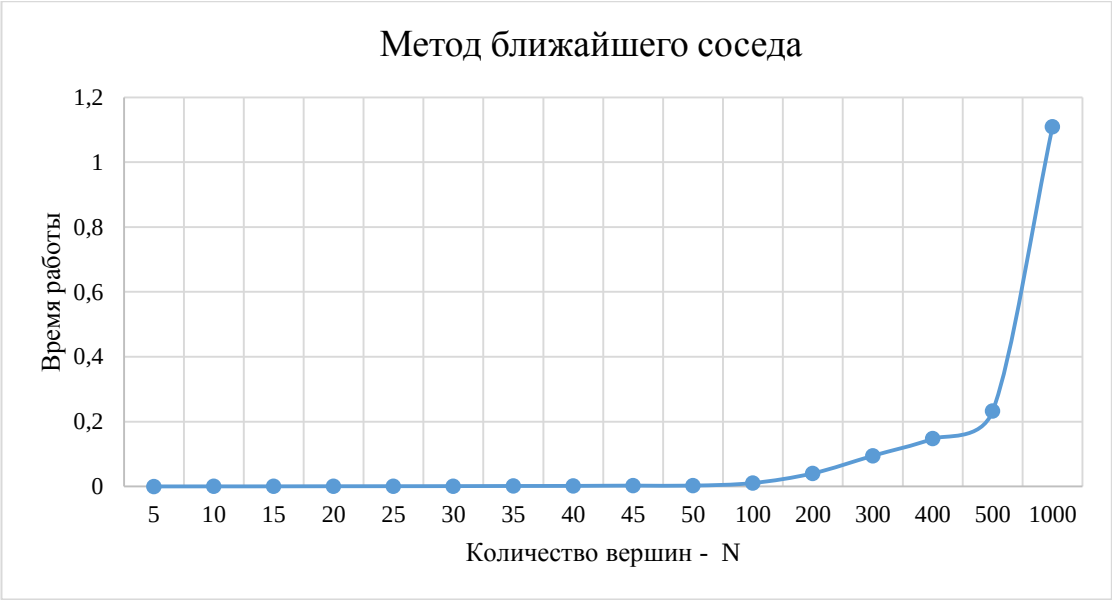

Рисунок 21 – График зависимости времени от количества N

На рисунках 22 и 23 представлены результаты расчета задачи коммивояжера с помощью метода ближайшего соседа при $N = 15$ и $N = 100$ соответственно.

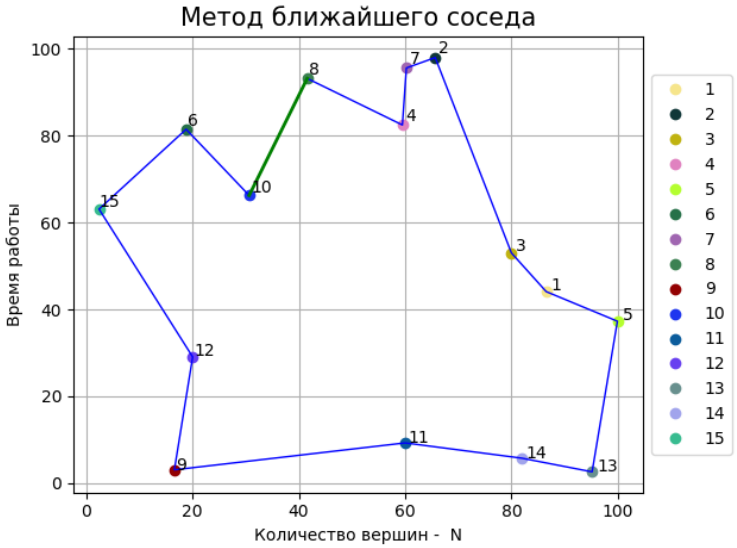

Рисунок 22 – Путь с помощью метода ближайшего соседа при $N = 15$

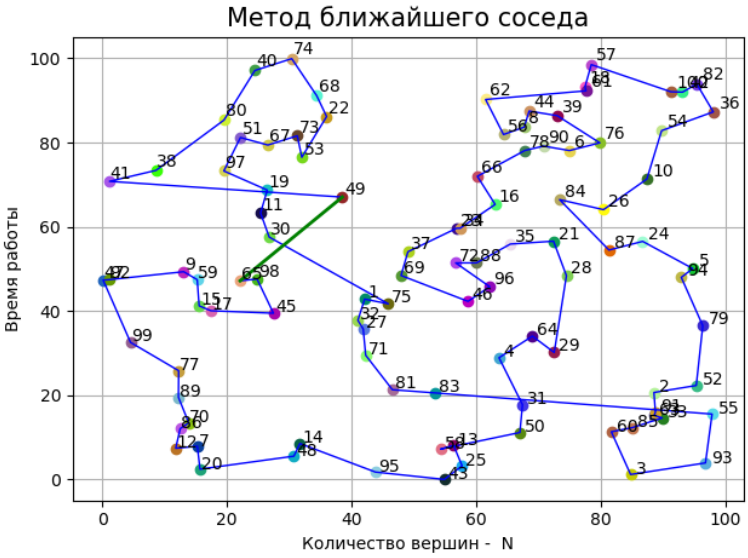

Рисунок 23 – Путь с помощью метода ближайшего соседа при $N = 100$

Исходя из результатов эксперимента, можно сделать вывод, что метод ближайшего соседа имеет большую скорость работы при малом количестве элементов.

3.3 Метод ветвей и границ

Во время работы метода ветвей и границ фиксируется время работы. Также программы выдает сумму всего построенного пути, номер вершины, с который был начат путь и конечную вершину тоже.

Исследования проводились до $N = 1000$, все данные представлены в таблице 17.

Таблица 17 – Результаты расчета маршрута

N	Метод ветвей и границ (миллисекунды)	Сумма всего пути
5	0,00051	211,42
10	0,0017	232,03
15	0,0048	340,86
20	0,00899	415,54
25	0,01637	439,64

Продолжение таблицы 17 – Результаты расчета маршрута

N	Метод ветвей и границ (миллисекунды)	Сумма всего пути
30	0,02671	443,44
35	0,04085	535,79
40	0,06163	477,7
45	0,08351	489,29
50	0,11281	650,47
100	0,84384	872,14
200	6,36346	1154,26
300	21,15128	1452,81
400	49,95794	1734,43
500	93,63642	2040,37
1000	206,4982	4126,58

На рисунке 24 представлена зависимость времени от количества N в матрице.

Рисунок 24 – График зависимости времени от количества N

На рисунках 25 и 26 представлены результаты расчета задачи коммивояжера с помощью метода ветвей и границ при $N = 15$ и $N = 100$ соответственно.

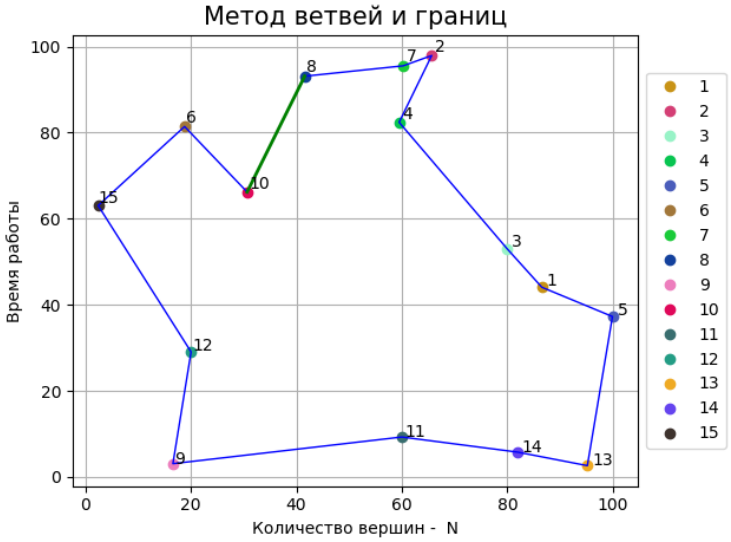

Рисунок 25 – Путь с помощью метода ветвей и границ при $N = 15$

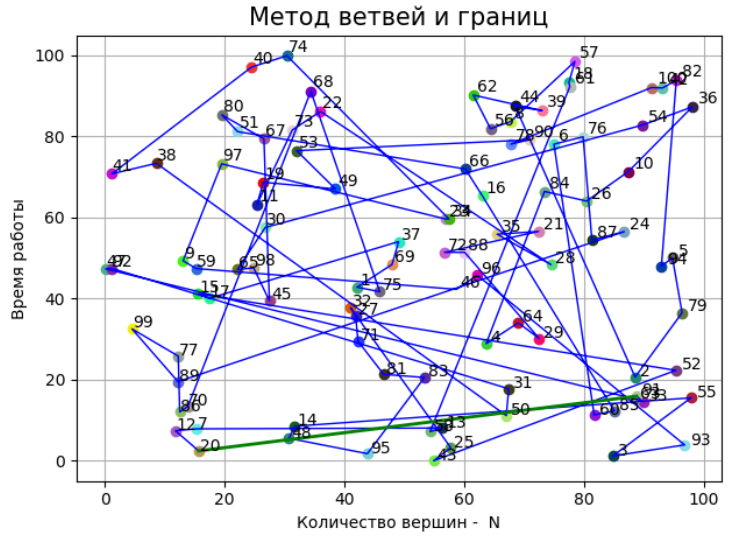

Рисунок 26 – Путь с помощью метода ветвей и границ при $N = 100$

Исходя из результатов эксперимента, можно сделать вывод, что время работы метода ветвей и границ на графике сильно возрастает с количеством элементов. Это подтверждается сложностью данного метода $O(n * log_2 n)$.

3.4 Сравнение двух реализованных алгоритмов

Сложность метода ближайшего соседа $O(n)$. Это означает, что при увеличении размера матрицы в два раза, затраты времени тоже возрастут в два раза.

Сложность метода ветвей и границ $O\ n * log_2 n$ – при увеличении матрицы в два раза, затраты на время также возрастают вдвое. Но сложность вычислений алгоритма увеличивается с ростом n. При малых значениях nалгоритмвносит большой вклад в затраты времени. Время начинает увеличиваться почти в четыре раза, но потом рост замедляется почти до линейного.

Объединим полученные результаты работы двух методов в одну таблицу (таблица 18).

Таблица 18 – Полученные результаты

N	Метод ближайшего соседа (миллисекунды)	Метод ветвей и границ (миллисекунды)
5	0,000059	0,00051
10	0,000227	0,0017
15	0,000315	0,0048
20	0,000680	0,00899
25	0,000773	0,01637
30	0,001077	0,02671
35	0,001437	0,04085
40	0,001748	0,06163
45	0,002623	0,08351
50	0,002625	0,11281
100	0,010576	0,84384
200	0,040370	6,36346
300	0,094620	21,15128

Продолжение таблицы 18 – Полученные результаты

N	Метод ближайшего соседа (миллисекунды)	Метод ветвей и границ (миллисекунды)
400	0,147421	49,95794
500	0,232296	93,63642
1000	1,109630	206,4982

Изобразим на графике объединение результатов двух методов для решения задачи коммивояжера (рисунок 27).

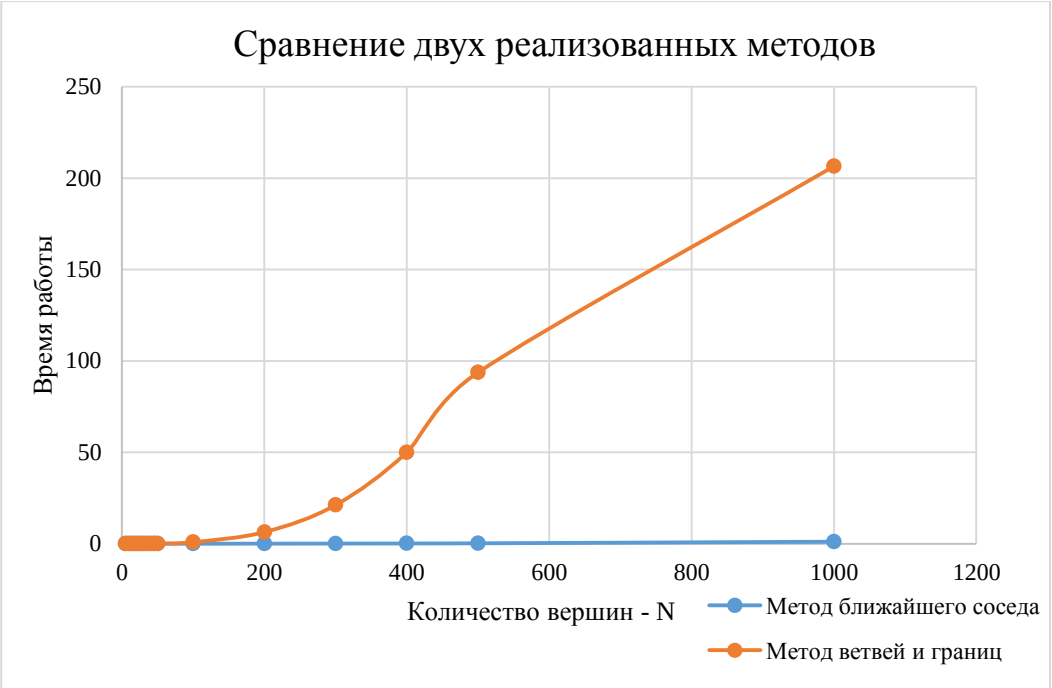

Рисунок 27 – Диаграмма сравнения реализованных методов

Из рисунка 27 явно видно, что метод ближайшего соседа работает намного быстрей, чем метод ветвей и границ.

Вычислим погрешность приближенного метода по сравнению с точным методом (таблица 19).

Таблица 19 – Вычисление погрешности

N	Сумма всего пути метода ветвей и границ (точный метод)	Сумма всего пути метода ближайшего соседа (приближенный метод)	Погрешность (%)
5	211,42	211,42	0
10	232,03	232,03	0
15	340,86	362,38	6, 313
20	415,54	421,46	1,425
25	439,64	460,23	4,683
30	443,44	514,65	16,059
35	523,14	510,7	2,378
40	477,7	540,16	13,075
45	489,29	564,79	15,431
50	650,47	669,89	2,986
100	872,14	888,41	1,866
200	1154,26	1233,23	6,842
300	1452,81	1505,75	3,644
400	1734,43	1748,69	0,822
500	1937,36	2040,37	5,317
1000	4126,58	2716,27	51,921

Посчитаем средний процент погрешности из таблицы 17, он будет равен 8,43 %. На основе среднего процента можно сказать, что в приближенном методе присутствует не значительное отклонение от точного метода [9].

Можно сделать вывод, что метод ближайшего соседа можно использовать в сферах вычисления больших количества городов, а метод ветвей и границ, наоборот, следует применять на небольших количествах.

ЗАКЛЮЧЕНИЕ

В ходе выполнения бакалаврской работы были рассмотрены существующие методы решения задачи коммивояжера, достоинства и недоставки всех методов, а также была произведена реализация метода ближайшего соседа и метода ветвей и границ на языке программирования Python.

В ходе работы были выполнены следующие задачи:

1. Рассмотрены существующие методы решения задачи коммивояжера;
2. Реализован метод ближайшего соседа для решения задачи коммивояжера на языке программирования Python;
3. Реализован метод ветвей и границ для решения задачи коммивояжера на языке программирования Python;
4. Выявлены достоинства каждого реализованного метода.
5. Проведены сравнения реализованных методов.

Были пошагово реализованы методы ближайшего соседа и ветвей и границ на языке программирования Python, с использованием пользовательского графического интерфейса.

Проведены вычислительные эксперименты, состоящие в сравнении времени работы данных методов, которые показали, что метод ближайшего соседа работает намного быстрее, чем метод ветвей и границ.

СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

Научная и методическая литература

1. Володина Е.В. Практическое применение алгоритма решения задачи коммивояжера/ Е.В. Володина, Е.А. Студентова // Инженерный Вестник Дона. – 2015. – №2-2. – 13 с.

2. Громкович Ю. Алгоритмизация труднорешаемых задач. Часть I. Простые примеры и простые эвристики / Ю. Громкович, Б.Ф.Мельников // Перевод с английского Б.Ф. Мельников.Философские проблемы информационных технологий и киберпространства. – Пятигорск: 2014, 360 с.

3. Громкович Ю. Алгоритмизация труднорешаемых задач. Часть II. Более сложные эвристики. / Ю. Громкович, Б.Ф.Мельников // Перевод с английского Б.Ф. Мельников.Философские проблемы информационных технологий и киберпространства. – Пятигорск: 2014, 612 с.

4. Гудман С. Введение в разработку и анализ алгоритмов: учебное пособие / С. Гудман, С. Хидетниеми //Перевод с английского Ю.Б. Котова, Л.В. Сухаревой, Л.В. Ухова под редакцией В.В. Мартынюка. – М: Мир, 1981. 368 с.

5. Дулькейт В.И. Приближённое решение задачи коммивояжера методов рекурсивного построения вспомогательной кривой / В.И. Дулькейт, Р. Т. Файзуллин // ПДМ. 2009. №1 (3). – 8 с.

6. Ермоленко С.В. Исследования решения задачи коммивояжера с помощью островной модели / С.В. Ермоленко, В. Г. Кобак // Электронный журнал «Молодой исследователь Дона». – Ростов-на-Дону. – 2016. – №3. – 7 с.

7. Ерзин А.И. Задачи маршрутизации: учеб. пособие / А.И. Ерзин, Ю.А. Кочетов. – Новосиб. гос ун-т. – Новосибирск: РИЦ НГУ, 2014. – 95 с.

8. Иваницкая А.В. Решение задачи маршрутизации в среде VBA / А.В.Иваницкая, Е.Н Едемская // 2 всеукраинская студенческая научно-техническая конференция «Современные информационные технологии в

образовании и научных исследованиях (СИТОНИ-2011) 13-14 октября 2011г. Сборник научных трудов студентов, магистров и преподавателей. – Донецк, 2011. – с.17 -22.

9. Кошевой Н.Д. Метод последовательного приближения длярешения задачи коммивояжера/ Н.Д. Кошевой, Е.М. Костенко,А. С. Чуйко // Математичнемоделювання. – 2012. – №1. – С. 58-61.

10. Ляликов В.Н. Комплекс программ для исследования методов решения задачи о коммивояжере диссертация кандидата технических наук / В.Н. Ляликов. – Ульяновск, 2008. – 123 с.

11. Макаркин С. Б. Подход к решению псевдогеометрической версии задачи коммивояжера / С. Б. Макаркин, Б.Ф. Мельников, М.А. Тренина // Научный журнал «Известия ВУЗов. Поволжский регион. Физико-математические науки.». – 2015. №2 (34). – с. 135 – 147.

12. Мельников Б. Ф. Кластеризация ситуаций в алгоритмах решения задачи коммивояжера и ее применение в некоторых прикладных задачах. Часть II. Списочная метрика и некоторые связанные оптимизационные проблемы / Б. Ф. Мельников, Е.В. Давыдова, Н.В. Ничипорчук,М.А. Тренина // Научный журнал «Известия ВУЗов. Поволжский регион. Физико-математические науки.». – 2018. №4 (48). – с. 62 – 77.

13. Романов П.С. Оптимизация параметров транспортного процесса на основе эвристических алгоритмов задачи коммивояжера / П.С. Романов, И.П. Романов // Электронный журнал «Интеллект. Инновации. Инвестиции». – 2018. – №1. – с. 67 – 71.

14. Семенов С. С. Анализ трудоемкости различных алгоритмических подходов для решения задачи коммивояжера/ С. С.Семенов, А.В. Педан, В.С. Воловиков, И.С. Климов // Системы управления, связи и безопасности. 2017. № 1. С. 116–131.

15. AhmedZ.H. The ordered clustered traveling salesman problem: a hybrid genetic algorithm // The Scientific World Journal. – 2014.

16. Korostensky C. Near optimal multiple sequence alignments using a travelling salesman problem approach / C. Korostensky, G. Gonnet // String Processing and Information Retrieval Symposium & International Workshop on Groupware. – 1999. P. 05 – 114.

17. Little J.D.C. An algorithm for the traveling salesmanproblem / J.D.C.Little, K.G. Murty, D.W. Sweeney, C Karel // Operations research. 1963, 11(6), 972 – 989.

18. MestriaM. New hybrid heuristic algorithm for the clustered traveling salesman problem. Computers & Industrial Engineering // 116: 1-12. –2018.

19. Padberg M.W. The travelling salesman problem and a class of polyhedra of diameter two / M.W. Padberg, M. R.Rao // Math. Program. –1974. –7(1), 32–45.

20. Potvin J.Y. A Genetic Algorithm for the Clustered Traveling Salesman Problem with a Prespecified Order on the Clusters / J.Y. Potvin, F. Guertin //Computer Science Interfaces Series. – vol 9. – Springer, Boston, MA, 1998.

Электронные ресурсы

21. Введение в оптимизацию. Имитация отжига [Электронный ресурс:] - Режим доступа: https://habr.com/ru/post/209610/

22. Метод отжига [Электронный ресурс:] - Режим доступа: http://smart-blog.net/post/1773

23. Презентация на тему: Задача коммивояжера [Электронный ресурс:] - Режим доступа: http://www.myshared.ru/slide/486789/

```
from tkinter import *
import matplotlib.pyplot as plt
import numpy as np
from numpy import exp, sqrt
import random
from timeit import default_timer as timer

clicks = 0


# Дерево решений
class Node:
    def __init__(self):
        self.value = 0
        self.next = 0
        self.root = 0
        self.way = 0

        self.rowNums = 0
        self.colNums = 0

        self.a = 0  # Координата задается рандмно от 0 до 100
        self.m = 0
        self.ib = 0  # номер первого города
        self.n = 0
        self.X = 0
        self.Y = 0

    def setDefault(self, n, a, m, ib, matrix, X, Y):
        self.a = a  # Координата задается рандомно от 0 до 100
        self.m = m
        self.ib = ib  # номер первого города
        self.n = n
        self.X = X
        self.Y = Y
        self.matrix = matrix

    def setValue(self, node):
        self.root = node
        self.rowNums = node.rowNums
        self.colNums = node.colNums
        self.matrix = node.matrix[:]

    def setNums(self):
        nums = [(x + 1) for x in range(len(self.matrix))]
        self.rowNums = nums[:]
        self.colNums = nums[:]


class BnP:
```

```
    @staticmethod
    def findSmallestElementInRow(array, row):
        """
        Поиск наименьшего значение в СТРОКЕ
        :param array:
        :param row:
        :return: минимальный элемент
        """
        minElement = sys.maxsize
        for element in array[row]:
            if minElement > element:
                minElement = element
        return minElement

    @staticmethod
    def findSmallestElementInCol(array, col):
        """
        Поиск наименьшего значение в СТОЛБЦЕ
        :param array:
        :param col:
        :return: минимальный элемент
        """
        minElement = sys.maxsize
        for element in array[:, col]:
            if minElement > element:
                minElement = element
        return minElement

    @staticmethod
    def rowReduction(array):
        """
        Редукция строк
        :param array:
        :return:
        """
        rowCoefficients = []
        for i in range(len(array)):
            minElement = BnP.findSmallestElementInRow(array, i)
            rowCoefficients.append(minElement)
            for j in range(len(array[i])):
                array[i][j] -= minElement
        return array, rowCoefficients

    @staticmethod
    def colReduction(array):
        """
        Редукция столбцов
        :param array:
        :return:
        """
        colCoefficients = []
        for i in range(len(array)):
```

```
            minElement = BnP.findSmallestElementInCol(array, i)
            colCoefficients.append(minElement)
            for j in range(len(array[:, i])):
                array[j][i] -= minElement
        return array, colCoefficients

    @staticmethod
    def evaluationCalculation(array):
        """
        Оценка нулевых точек (коэффициент по строке и столбцу)
        :param array:
        :return:
        """
        maxAssessment = -1  # Максимальная оценка
        maxI = -1  # Строка максимального значения
        maxJ = -1  # Столбец максимального значения

        for i in range(len(array)):
            for j in range(len(array[i])):
                if array[i][j] == 0:
                    minRow = sys.maxsize
                    minCol = sys.maxsize

                    for k in range(len(array[i])):
                        if array[i][k] < minRow and k != j:
                            minRow = array[i][k]

                    for k in range(len(array[:, j])):
                        if array[k][j] < minCol and k != i:
                            minCol = array[k][j]

                    sumOfElements = minCol + minRow  # Сумма
элементов по столбцу и строке

                    if sumOfElements > maxAssessment:
                        maxAssessment = sumOfElements
                        maxI = i
                        maxJ = j

        return maxI, maxJ

    @staticmethod
    def cutMatrix(array, indexI, indexJ):
        """
        Обрезаем матрицу по индексам i и j
        :param array:
        :param indexI:
        :param indexJ:
        :return:
        """
        newArray = []
```

```
        for i in range(len(array)):
            if i != indexI:
                newRow = []
                for j in range(len(array)):
                    if j != indexJ:
                        newRow.append(array[i][j])
                newArray.append(newRow)

        return np.array(newArray)

    @staticmethod
    def startIteration(node):
        """
        Проводим одну итерацию:
            > находим минимум по строкам и столбцам
            > считаем полный путь, учитывая путь предыдущего
значения
            > находим ноль с максимальной оценкой
                > задаём все пути к нему равным бесконечности
                > удаляем сам путь
            > заносим в массив путей очередной путь
        :param node:
        :return:
        """
        node.matrix, sumOfRowReduction =
BnP.rowReduction(node.matrix)
        node.matrix, sumOfColReduction =
BnP.colReduction(node.matrix)
        node.value = sum(sumOfColReduction) +
sum(sumOfRowReduction)

        if node.root != 0:
            node.value += node.root.value

        indexI, indexJ = BnP.evaluationCalculation(node.matrix)
        node.way = [node.rowNums[indexI], node.colNums[indexJ]]

        node.next = Node()
        node.next.setValue(node)

        node.next.matrix[indexI][indexJ] = float("inf")
        node.next.matrix = BnP.cutMatrix(node.matrix, indexI,
indexJ)

        del node.next.rowNums[indexI]
        del node.next.colNums[indexJ]

        if (node.way[0] in node.next.colNums) and (node.way[1]
in node.next.rowNums):
            indexI = node.next.rowNums.index(node.way[1])
            indexJ = node.next.colNums.index(node.way[0])
            node.next.matrix[indexI][indexJ] = float("inf")
```

```
        if len(node.matrix) > 1:
            BnP.startIteration(node.next)

    @staticmethod
    def getResult(node):
        """
        Выдать результат всего пути
        :param node:
        :return:
        """
        if node.next != 0 and len(node.next.matrix) != 1:
            return BnP.getResult(node.next)
        else:
            return node.value

    @staticmethod
    def getWayNode(node, way):
        """
        Сбор всего пути
        :param node:
        :param way:
        :return:
        """
        way.append(node.way)
        if node.next != 0 and len(node.next.matrix) != 0:
            way = BnP.getWayNode(node.next, way)

        return way

    @staticmethod
    def setResultWay(array):
        resultWay = array[0]

        for i in range(len(array[1])):
            for j in range(2):
                set = array[1][i][j]
                check = True
                for k in range(len(resultWay)):
                    if resultWay[k] == set:
                        check = False
                if check:
                    resultWay.insert(len(resultWay) - 1, set)
        return resultWay

    @staticmethod
    def getResultWay(node):
        """
        Получить вектор опитмального пути
        :param node:
        :return:
        """
```

```
        allWays = BnP.getWayNode(node, [])
        optimalWay = []

        for z in range(len(allWays)):
            fullWay = allWays[:]
            optimalWay = [fullWay[z][0], fullWay[z][1]]
            del fullWay[z]

            isCorrect = True
            while len(fullWay) != 0 and isCorrect:
                flag = False

                for l in range(len(fullWay)):
                    bunch = fullWay[l]
                    if bunch[0] == optimalWay[(len(optimalWay) -
1)]:
                        optimalWay.append(bunch[1])
                        del fullWay[l]

                        flag = True
                        break

                if not flag:
                    isCorrect = False

        return [optimalWay, allWays]


class NearestNeighbor:
    @staticmethod
    def hex_code_colors():
        a = hex(random.randrange(0, 256))
        b = hex(random.randrange(0, 256))
        c = hex(random.randrange(0, 256))
        a = a[2:]
        b = b[2:]
        c = c[2:]
        if len(a) < 2:
            a = "0" + a
        if len(b) < 2:
            b = "0" + b
        if len(c) < 2:
            c = "0" + c
        z = a + b + c
        return "#" + z.upper()

    @staticmethod
    def methodImplementation(self, index):
        w = []
        M = self.matrix.copy()
        w.append(index)
        for i in np.arange(1, self.n, 1):
```

```
            s = []
            for j in np.arange(0, self.n, 1):
                s.append(M[w[i-1], j])
            w.append(s.index(min(s)))
            for j in np.arange(0, i, 1):
                M[w[i], w[j]] = float('inf')
                M[w[i], w[j]] = float('inf')
        S = sum([sqrt((self.X[w[i]]-
self.X[w[i+1]])**2+(self.Y[w[i]]-self.Y[w[i+1]])**2)
                 for i in np.arange(0, self.n - 1, 1)]) +
sqrt((self.X[w[self.n-1]]-self.X[w[0]])**2+(self.Y[w[self.n-1]]-
self.Y[w[0]])**2)
        return S, w

    def __init__(self, n, a, m, ib, matrix, X, Y):
        self.a = a  # Координата задается рандмно от 0 до 100
        self.m = m
        self.ib = ib  # номер первого города
        self.n = n
        self.X = X
        self.Y = Y
        self.matrix = matrix

    @staticmethod
    def runMethod(self):
        way = []  # путь прохода

        Matrix = np.zeros([self.n, self.n])  # Шаблон матрицы
относительных расстояний между пунктами
        for i in np.arange(0, self.n, 1):
            for j in np.arange(0, self.n, 1):
                if i != j:
                    Matrix[i, j] = sqrt((self.X[i]-
self.X[j])**2+(self.Y[i]-self.Y[j])**2)  # Заполнение матрицы
                else:
                    Matrix[i, j] = float('inf')  # Заполнение
главной диагонали матрицы

        Summs = []
        Ways = []

        for i in range(self.n):
            start_time = timer()
            tempS, tempWay =
NearestNeighbor.methodImplementation(self, i)
            end_time = timer() - start_time
            Summs.append(tempS)
            Ways.append(tempWay)

        indexWay = Summs.index(min(Summs))
        way = Ways[indexWay]
```

```
        time.config(text="Время работы алгоритма %f" %
float(end_time))
        summs.config(text="Сумма маршрута %s" %
round(Summs[indexWay], 2))

        plt.title("Метод ближайшего соседа", size=15)
        X1 = [self.X[way[i]] for i in np.arange(0, self.n, 1)]
        Y1 = [self.Y[way[i]] for i in np.arange(0, self.n, 1)]


        # Рандомные цвета
        t = []
        for i in range(self.n):
            plt.plot(self.X[i], self.Y[i],
color=NearestNeighbor.hex_code_colors(), linestyle=' ',
marker='o')
            t.append(i + 1)
            plt.text(self.X[i] * (1 + 0.01), self.Y[i] * (1 +
0.01), i + 1)

        # Обозначение каждой точки
        if len(t) < 16:
            plt.legend(t, loc='center left', bbox_to_anchor=(1.,
0.5), ncol=1)

        plt.plot(X1, Y1, color='b', linewidth=1)
        X2 = [self.X[way[self.n-1]], self.X[way[0]]]
        Y2 = [self.Y[way[self.n-1]], self.Y[way[0]]]
        plt.xlabel("Количество вершин -  N")
        plt.ylabel("Время работы")
        plt.plot(X2, Y2, color='g', linewidth=2,  linestyle='-')
        plt.grid(True)
        plt.show()


class MyMatrix:
    def __init__(self):
        self.matrix = 0
        self.beforeN = 0
        self.X = 0
        self.Y = 0

    def generateMatrix(self, n):
        a = 0  # Координата задается рандмно от 0 до 100
        m = 100
        ib = 0  # номер первого города
        way = []  # путь прохода

        self.X = np.random.uniform(a, m, n)
        self.Y = np.random.uniform(a, m, n)
```

```
        Matrix = np.zeros([n, n])  # Шаблон матрицы
относительных расстояний между пунктами
        for i in np.arange(0, n, 1):
            for j in np.arange(0, n, 1):
                if i != j:
                    Matrix[i, j] = sqrt((self.X[i] - self.X[j])
** 2 + (self.Y[i] - self.Y[j]) ** 2)  # Заполнение матрицы
                else:
                    Matrix[i, j] = float('inf')  # Заполнение
главной диагонали матрицы
        self.matrix = Matrix
        self.beforeN = n


M = MyMatrix()


def runNearestNeighbor():
    n = town.get()
    if not n.isdigit():
        cheak['text'] = " " * 80
        cheak['text'] = "Введите число"
    elif int(n) <= 1:
        cheak['text'] = " " * 80
        cheak['text'] = "Количество точек должно быть больше 1"
    else:
        cheak['text'] = " " * 80
        n = int(n)
        if M.beforeN == 0 or M.beforeN != n:
            M.generateMatrix(n)

        methodNeighbor = NearestNeighbor(n, 0, 100, 0, M.matrix,
M.X, M.Y)
        NearestNeighbor.runMethod(methodNeighbor)


def runBnP():
    n = town.get()
    if not n.isdigit():
        cheak['text'] = " " * 80
        cheak['text'] = "Введите число"
    elif int(n) <= 1:
        cheak['text'] = " " * 80
        cheak['text'] = "Количество точек должно быть больше 1"
    else:
        cheak['text'] = " " * 80
        n = int(n)
        if M.beforeN == 0 or M.beforeN != n:
            M.generateMatrix(n)

        rootNode = Node()
        rootNode.setDefault(n, 0, 100, 0, M.matrix, M.X, M.Y)
```

```
        rootNode.setNums()

        matrixSize = len(rootNode.matrix)
        start_time = timer()
        BnP.startIteration(rootNode)
        time.config(text="Время работы алгоритма %s" %
round((timer() - start_time), 5))

        way = BnP.getResultWay(rootNode)[0]

        if len(way) != matrixSize + 1:
            way = BnP.setResultWay(BnP.getResultWay(rootNode))
        del way[-1]
        way = [x - 1 for x in way]

        summs.config(text="Сумма маршрута %s" %
round(BnP.getResult(rootNode), 2))
        plt.title("Метод ветвей и границ", size=15)
        X1 = [rootNode.X[way[i]] for i in np.arange(0, n, 1)]
        Y1 = [rootNode.Y[way[i]] for i in np.arange(0, n, 1)]
        # Рандомные цвета
        t = []
        for i in range(n):
            plt.plot(rootNode.X[i], rootNode.Y[i],
color=NearestNeighbor.hex_code_colors(), linestyle=' ',
marker='o')
            t.append(i + 1)
            plt.text(rootNode.X[i] * (1 + 0.01), rootNode.Y[i] *
(1 + 0.01), i + 1)

        # Обозначение каждой точки
        if len(t) < 16:
            plt.legend(t, loc='center left', bbox_to_anchor=(1.,
0.5), ncol=1)

        plt.plot(X1, Y1, color='b', linewidth=1)
        X2 = [rootNode.X[way[rootNode.n - 1]],
rootNode.X[way[0]]]
        Y2 = [rootNode.Y[way[rootNode.n - 1]],
rootNode.Y[way[0]]]
        plt.plot(X2, Y2, color='g', linewidth=2, linestyle='-')
        plt.xlabel("Количество вершин -  N")
        plt.ylabel("Время работы")
        plt.grid(True)
        plt.show()


def creatMatrix():
    n = town.get()
    if not n.isdigit():
        cheak['text'] = " " * 80
        cheak['text'] = "Введите число"
```

```
    elif int(n) <= 1:
        cheak['text'] = " " * 80
        cheak['text'] = "Количество точек должно быть больше 1"
    else:
        cheak['text'] = " " * 80
        n = int(n)
        M.generateMatrix(n)


#  цвет
bg = '#e7f2e1'
#  создание окна интерфейса
root = Tk()
root.configure(background=bg)
root.title("Решение задачи коммивояжера")
root.geometry("720x200")

town = StringVar()
town_label = Label(text="Введите количество городов:",
fg="#000000", bg=bg, font='Times 14')
town_label.grid(row=0, column=0, sticky="w")

town_entry = Entry(textvariable=town)
town_entry.grid(row=0, column=1, padx=5, pady=5)

#  кнопка для решения задачи коммивояжера методом ближайшего
соседа
message_button = Button(text="Nearest neighbor ", fg="#000000",
                        bg="#c6f2ae", font='Times 14',
command=runNearestNeighbor)
message_button.grid(row=0, column=2, padx=5, pady=5, sticky="w")
#  кнопка для решения задачи коммивояжера методом ветвей и
границ
message_button2 = Button(text="BnP", fg="#000000",
                         bg="#c6f2ae", font='Times 14',
command=runBnP)
message_button2.grid(row=0, column=3, padx=5, pady=5,
sticky="w")
#  кнопка для создания матрциы
matrix_button = Button(text="Create matrix", fg="#000000",
bg="#c6f2ae", font='Times 14', command=creatMatrix)
matrix_button.grid(row=0, column=4, padx=5, pady=5, sticky="w")

#  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
cheak = Label(font='Times 14', bg=bg, fg='red')
cheak.grid(row=5, column=0, sticky="w", columnspan=3)
root.mainloop()
```