Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Отчет по лабораторной работе №6 по курсу "Анализ алгоритмов"

Тема Муравьиный алгоритм

Студент Зайцева А.А.

Группа ИУ7-52Б

Преподаватели Волкова Л.Л.

Москва — 2021 г.

Оглавление

Введение

Муравьиные алгоритмы представляют собой новый перспективный метод решения задач оптимизации, в основе которого лежит моделирование поведения колонии муравьев. Колония представляет собой систему с очень простыми правилами автономного поведения особей. Однако, несмотря на примитивность поведения каждого отдельного муравья, поведение всей колонии оказывается достаточно разумным. Эти принципы проверены временем — удачная адаптация к окружающему миру на протяжении миллионов лет означает, что природа выработала очень удачный механизм поведения [1].

Целью данной работы является реализация муравьиного алгоритма для решения задачи коммивояжера и приобретение навыков параметризации алгоритмов.

В рамках выполнения работы необходимо решить следующие задачи:

1) реализовать алгоритм полного перебора для решения задачи коммивояжера;

2) изучить и реализовать муравьиный алгоритм для решения задачи коммивояжера;

3) провести параметризацию муравьиного алгоритма на трех классах данных и подобрать оптимальные параметры;

4) провести сравнительный анализ времени выполнения и трудоемкостей реализаций.

1 Аналитическая часть

В данном разделе будет приведена теория, необходимая для разработки и реализации двух алгоритмов решения задачи коммивояжера: алгоритма полного перебора и муравьиного алгоритма.

1.1 Задача коммивояжера

В задаче коммивояжера рассматривается n городов и матрица попарных расстояний между ними. Требуется найти такой порядок посещения городов, чтобы суммарное пройденное расстояние было минимальным, каждый город посещался ровно один раз, и коммивояжер вернулся в тот город, с которого начал свой маршрут. Другими словами, во взвешенном полном графе требуется найти гамильтонов цикл минимального веса [2].

1.2 Алгоритм полного перебора для решения задачи коммивояжера

Суть алгоритма полного перебора для решения задачи коммивояжера заключается в переборе всех вариантов путей и нахождении кратчайшего. Преимуществом является его результат - точное решение, недостатком - длительность вычислений при относительно небольшом пространстве поиска [3]. Эта задача является NP-трудной, и точный переборный алгоритм ее решения имеет факториальную сложность [1].

1.3 Муравьиный алгоритм для решения задачи коммивояжера

Моделирование поведения муравьев связано с распределением феромона на тропе — ребре графа в задаче коммивояжера. При этом вероятность

включения ребра в маршрут отдельного муравья пропорциональна количеству феромона на этом ребре, а количество откладываемого феромона пропорционально длине маршрута. Чем короче маршрут, тем больше феромона будет отложено на его ребрах. При этом избежать преждевременной сходимости можно, моделируя отрицательную обратную связь в виде испарения феромона. С учетом особенностей задачи коммивояжера, мы можем описать локальные правила поведения муравьев при выборе пути.

- Муравьи имеют собственную «память» в виде списка уже посещенных городов. Обозначим через $J_{i,k}$ список городов, которые необходимо посетить муравью k, находящемуся в городе i.

- Муравьи обладают «зрением» — видимость есть эвристическое желание посетить город j, если муравей находится в городе i. Будем считать, что видимость обратно пропорциональна расстоянию между городами i и j — D_{ij}: $\eta_{ij} = \frac{1}{D_{ij}}$.

- Муравьи обладают «обонянием» — они могут улавливать след феромона, подтверждающий желание посетить город j из города i, на основании опыта других муравьев. Количество феромона на ребре (i, j) в момент времени t обозначим через $\tau_{ij}(t)$.

На этом основании мы можем сформулировать вероятностно-пропорционально правило 1.1, определяющее вероятность перехода k-ого муравья из города i в город j:

$$P_{ij,k}(t) = \begin{cases} \frac{(\tau_{ij}(t))^{\alpha}(\eta_{ij}(t))^{\beta}}{\sum\limits_{l \in J_{i,k}} (\tau_{il}(t))^{\alpha}(\eta_{il}(t))^{\beta}}, j \in J_{i,k}, \\ 0, \text{иначе} \end{cases}, \tag{1.1}$$

где $\alpha-$ параметр влияния длины пути, $\beta-$ параметр влияния феромона. При $\alpha = 0$ будет выбран ближайший город, что соответствует жадному алгоритму в классической теории оптимизации. Если $\beta = 0$ работает лишь феромонное усиление, что влечет за собой быстрое вырождение маршрутов к одному субоптимальному решению [2].

Пройдя ребро (i, j), муравей откладывает на нем некоторое количество феромона, которое должно быть связано с оптимальностью сделанного вы-

бора. Пусть $T_k(t)$ есть маршрут, пройденный муравьем k к моменту времени t , а $L_k(t)$ — длина этого маршрута. Пусть также Q — параметр, имеющий значение порядка длины оптимального пути. Тогда откладываемое количество феромона может быть задано в виде 1.2:

$$\Delta\tau_{ij,k}(t) = \begin{cases} \frac{Q}{L_k(t)}, (i,j) \in T_k(t), \\ 0, \text{иначе.} \end{cases} \tag{1.2}$$

Правила внешней среды определяют, в первую очередь, испарение феромона. Пусть $p \in [0,1]$ есть коэффициент испарения, тогда правило испарения имеет вид 1.3:

$$\tau_{ij}(t+1) = (1-p)\tau_{ij}(t) + \Delta\tau_{ij}(t), \Delta\tau_{ij} = \sum_{k=1}^{m} \Delta\tau_{ij,k}(t), \tag{1.3}$$

где m — количество муравьев в колонии.

В начале алгоритма количество феромона на ребрах принимается равным небольшому положительному числу. При этом необходимо следить, чтобы количество феромона на существующем ребре не обнулилось в ходе испарения. Общее количество муравьев остается постоянным и равным количеству городов, каждый муравей начинает маршрут из своего города.

1.4 Вывод из аналитической части

Были рассмотрены идеи и материалы, необходимые для разработки и реализации двух алгоритмов решения задачи коммивояжера: алгоритма полного перебора и муравьиного алгоритма.

2 Конструкторская часть

В данном разделе будут приведены схемы алгоритма полного перебора и муравьиного алгоритма для решения задачи коммивояжера.

2.1 Схема алгоритма полного перебора

На рисунке 2.1 приведена схема вспомогательной функции next_set, которая позволяет сгенерировать все возможные перестановки первых n неотрицательных чисел.

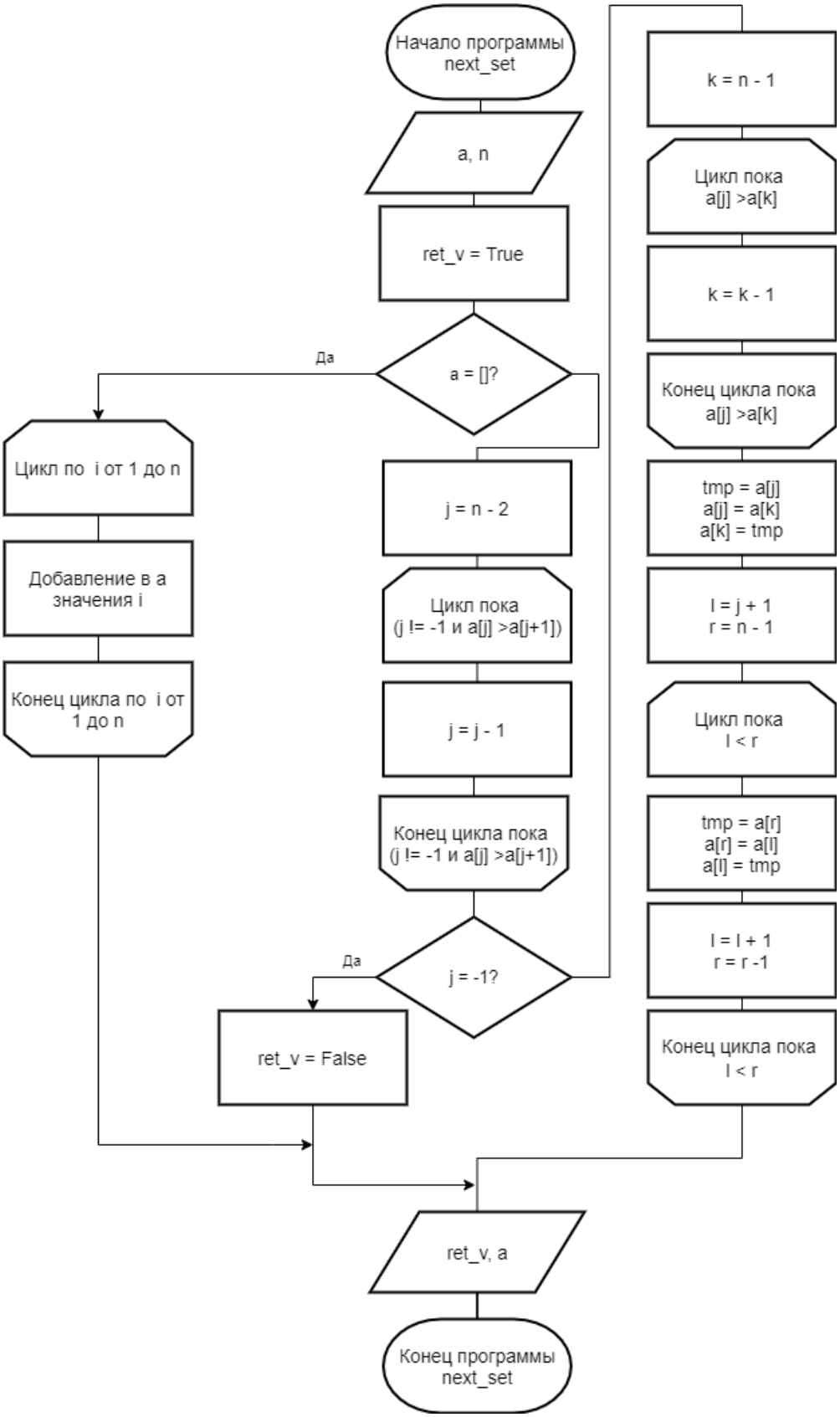

Рисунок 2.1 – Схема вспомогательной функции next_set

На рисунке 2.2 приведена схема вспомогательной функции count_way_len, которая с помощью матрицы смежности D вычисляет длину пути при проходе по городам visited_cities.

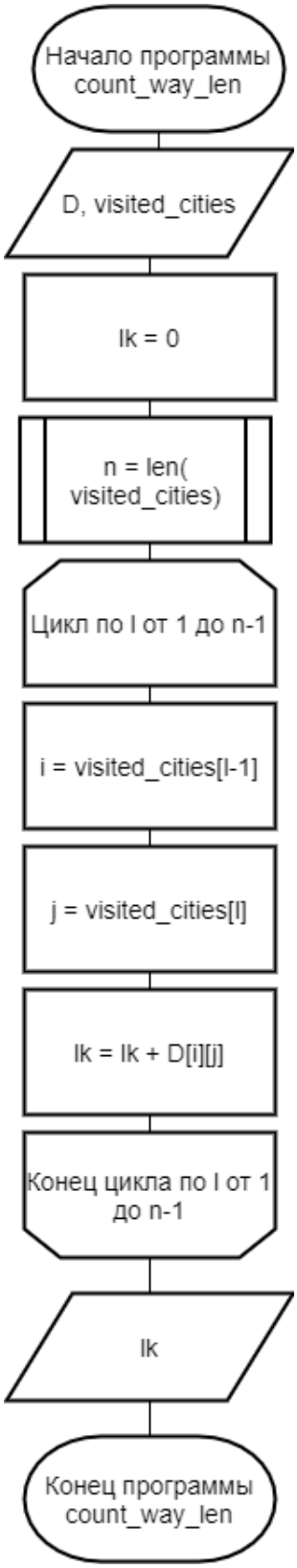

Рисунок 2.2 – Схема вспомогательной функции count_way_len

На рисунке 2.3 приведена схема алгоритма полного перебора для решения задачи коммивояжера.

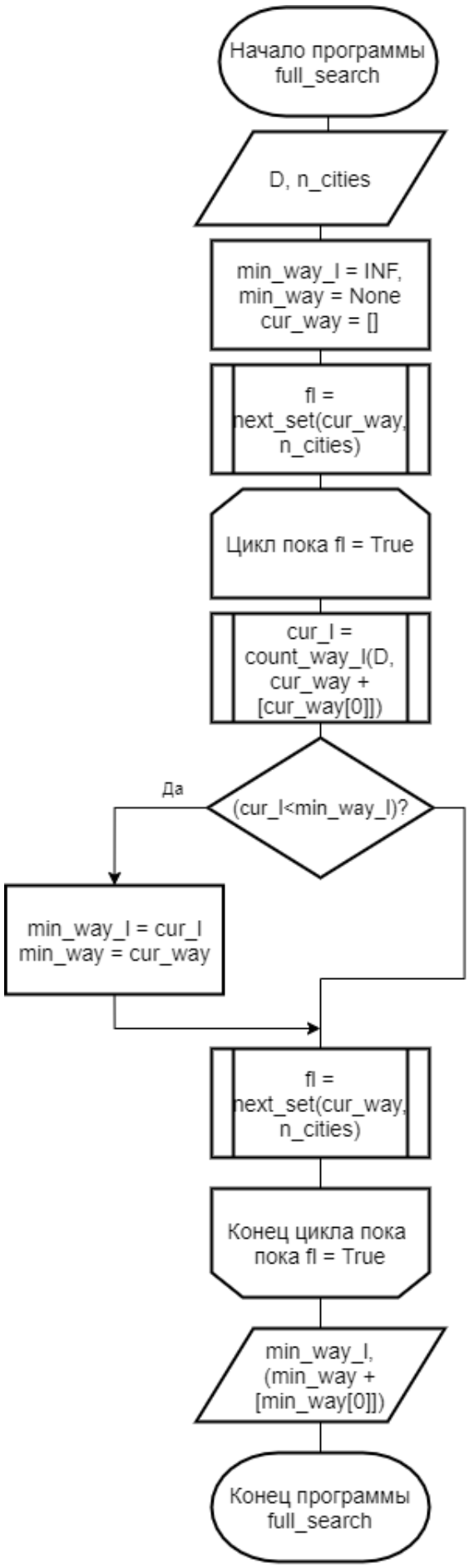

Рисунок 2.3 – Схема алгоритма полного перебора

2.2 Схема муравьиного алгоритма

На рисунке 2.4 приведена схема муравьиного алгоритма для решения задачи коммивояжера.

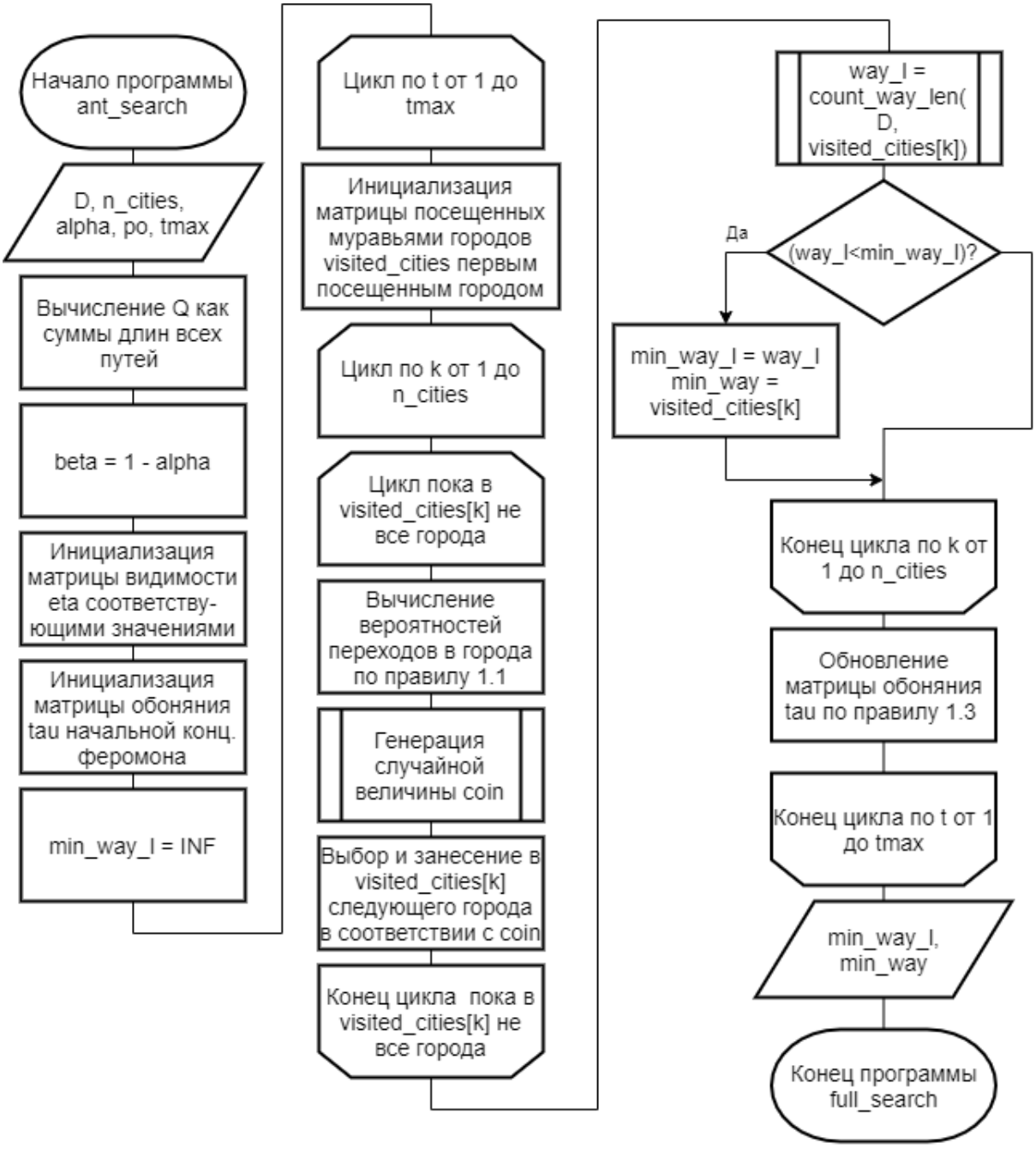

Рисунок 2.4 – Схема муравьиного алгоритма

2.3 Вывод из конструкторской части

Были приведены схемы разрабатываемых алгоритмов.

3 Технологическая часть

В данном разделе производится выбор средств реализации, приводятся требования к ПО, листинги реализованных алгоритмов решения задачи коммивояжера, а также результаты их тестированиия.

3.1 Требования к ПО

На вход программе подаются количество городов и симметричная матрица смежности, а также параметры α (коэффициент жадности), ро (коэффициент испарения) и tmax (время жизни колонии) для муравьиного алгоритма. На выходе должны быть получены решения задачи коммивояжера, найденые с помощью каждого реализованного алгоритма: алгоритма полного перебора и муравьиного алгоритма.

3.2 Выбор средств реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язык Python [4]. Он позволяет быстро реализовывать различные алгоритмы без выделения большого времени на проектирование сруктуры программы и выбор типов данных.

Кроме того, в Python есть библиотека time, которая предоставляет функцию process_time для замера процессорного времени [5].

В качестве среды разработки выбран PyCharm. Он является кроссплатформенным, а также предоставляет удобный и функциональнаый отладчик и средства для рефакторинга кода, что позволяет быстро находить и исправлять ошибки [6].

3.3 Листинги кода

В листинге 3.1 представлена реализация алгоритма полного перебора для решения задачи коммивояжера.

Листинг 3.1 – Алгоритм полного перебора

```
def full_search(D, n_cities):
    min_way_length = INF
    min_way = None
    cur_way = []

    while next_set(cur_way, n_cities):
        cur_way_lenth = count_way_lenth(D, cur_way + [cur_way[0], ])
        if cur_way_lenth < min_way_length:
            min_way_length = cur_way_lenth
            min_way = cur_way

    return min_way_length, min_way + [min_way[0]]
```

В листинге 3.2 представлена реализация муравьиного алгоритма для решения задачи коммивояжера.

Листинг 3.2 – Муравьиный алгоритм

```
def ant_search(D, n_cities, alpha=ALPHA, po=PO, tmax=TMAX):
  Q = 0
  for i in range(n_cities):
    for j in range(i):
      if D[i][j] < INF:
        Q += D[i][j]
  beta = 1 - alpha

  eta = [[0 for i in range(n_cities)] for j in range(n_cities)]
  tau = [[0 for i in range(n_cities)] for j in range(n_cities)]
  for i in range(n_cities):
    for j in range(i):
      eta[i][j] = 1 / D[i][j]
      eta[j][i] = 1 / D[j][i]
      tau[i][j] = 2 * EPS
      tau[j][i] = 2 * EPS

  min_way_length = INF


  for t in range(tmax):
    visited_cities = [[i] for i in range(n_cities)]

    for k in range(n_cities):

      while len(visited_cities[k]) != n_cities:
        P_ch = [0 for i in range(n_cities)]
        for j in range(n_cities):
          if j not in visited_cities[k]:
            i = visited_cities[k][-1]
            P_ch[j] = (tau[i][j] ** alpha) * (eta[i][j] ** beta)

        P_zn = sum(P_ch)
        for j in range(n_cities):
          P_ch[j] /= P_zn

        coin = random()
        summ, j = 0, 0
```

```
            while summ < coin:
                summ += P_ch[j]
                j += 1
            visited_cities[k].append(j - 1)

        visited_cities[k].append(visited_cities[k][0])
        way_length = count_way_lenth(D, visited_cities[k])


        if way_length < min_way_length:
            min_way_length = way_length
            min_way = visited_cities[k]


    for i in range(n_cities):
        for j in range(i):
            delta_tau = 0

            for k in range(n_cities):
                way_length = count_way_lenth(D, visited_cities[k])
                for m in range(1, len(visited_cities[k])):
                    if (visited_cities[k][m], visited_cities[k][m - 1])
                        in ((i, j), (j, i)):
                        delta_tau += Q / way_length
                        break

        tau[i][j] = tau[i][j] * (1 - po) + delta_tau
        if tau[i][j] < EPS:
            tau[i][j] = EPS

    return min_way_length, min_way
```

В листинге 3.3 представлены реализации вспомогательных функций.

Листинг 3.3 – Вспомогательные функции

```
1
2  def next_set(a, n):
3      if not a:
4          for i in range(n):
5              a.append(i)
6          return True
7
8      j = n - 2
9      while j != -1 and a[j] > a[j + 1]:
10         j -= 1
11     if j == -1:
12         return False
13
14     k = n - 1
15     while a[j] > a[k]:
16         k -= 1
17     a[j], a[k] = a[k], a[j]
18
19     l = j + 1
20     r = n - 1
21     while l < r:
22         a[l], a[r] = a[r], a[l]
23         l += 1
24         r -= 1
25     return True
26
27 def count_way_lenth(D, visited_cities):
28     lk = 0
29     for l in range(1, len(visited_cities)):
30         i = visited_cities[l - 1]
31         j = visited_cities[l]
32         lk += D[i][j]
33
34     return lk
```

3.4 Тестирование

В таблице 3.1 приведены функциональные тесты для алгоритма решения задачи коммивояжера полным перебором. Все тесты пройдены успешно этим каждым алгоритмом.

Таблица 3.1 – Тестирование функций

Матрица смежности	Ожидаемый наименьший путь
$\begin{pmatrix} 0 & 3 & 4 & 7 \\ 3 & 0 & 3 & 7 \\ 4 & 3 & 0 & 7 \\ 7 & 7 & 7 & 0 \end{pmatrix}$	20, [0, 1, 2, 3, 0]
$\begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}$	6, [0, 1, 2, 3, 4, 5, 0]
$\begin{pmatrix} 0 & 1 & INF & 4 \\ 1 & 0 & 2 & INF \\ INF & 2 & 0 & 3 \\ 4 & INF & 3 & 0 \end{pmatrix}$	10, [0, 1, 2, 3, 0]

Так как муравьиный алгоритм не гарантирует получение точного результата, для его тестирования будем отслеживать выполнение отдельных его этапов. Выберем 'отладочного муравья' и 'отладочную итерацию'. Для муравья будем отслеживать вычисление вероятностей посещения городов, а также конечный выбор очередного города. По итерациям будем отслеживать изменение матрицы феромонов и процесс обновления наиболее оптимального пути. На рисунках 3.1- 3.5 приведены результаты тестирования, по которым видно, что ход выполнения программы верный.

```
Матрица:
  0   1 INF   4
  1   0   2 INF
INF   2   0   3
  4 INF   3   0
Ожидаемый результат:
 10, [0, 1, 2, 3, 0]

Результат алгоритма полного перебора:
 10, [3, 2, 1, 0, 3]
Длина минимального пути совпала с ожидаемой? - True

Муравьиный алгоритм


Отладочный муравей 2 на 0-й итерации
    Поиск 2-го города для посещения после посещения городов [2]
        Подсчитанные вероятности посещения городов: [7.785330107978534e-06, 0.5505059713127417, 0.0, 0.4494862433571504]
        Результат "подброса монетки": 0.28428923590999877
        Выбранный город": 1
        Полученный список посещенных городов": [2, 1]
```

Рисунок 3.1 – Результаты тестирования муравьиного алгоритма, часть 1

```
Отладочный муравей 2 на 0-й итерации
    Поиск 3-го города для посещения после посещения городов [2, 1]
        Подсчитанные вероятности посещения городов: [0.9999900000099999, 0.0, 0.0, 9.999900000999992e-06]
        Результат "подброса монетки": 0.3126929009346626
        Выбранный город": 0
        Полученный список посещенных городов": [2, 1, 0]


Отладочный муравей 2 на 0-й итерации
    Поиск 4-го города для посещения после посещения городов [2, 1, 0]
        Подсчитанные вероятности посещения городов: [0.0, 0.0, 0.0, 1.0]
        Результат "подброса монетки": 0.3612857546986338
        Выбранный город": 3
        Полученный список посещенных городов": [2, 1, 0, 3]
Итоговый результат отладочного муравья:
    список посещенных городов: [2, 1, 0, 3, 2]; длина пути: 10


После 0-й отладочной итерации
    Матрица следов феромонов:
  0   4   0   4
  4   0   4   0
  0   4   0   4
  4   0   4   0
Наиболее оптимальный путь: [0, 1, 2, 3, 0] длиной 10
```

Рисунок 3.2 – Результаты тестирования муравьиного алгоритма, часть 2

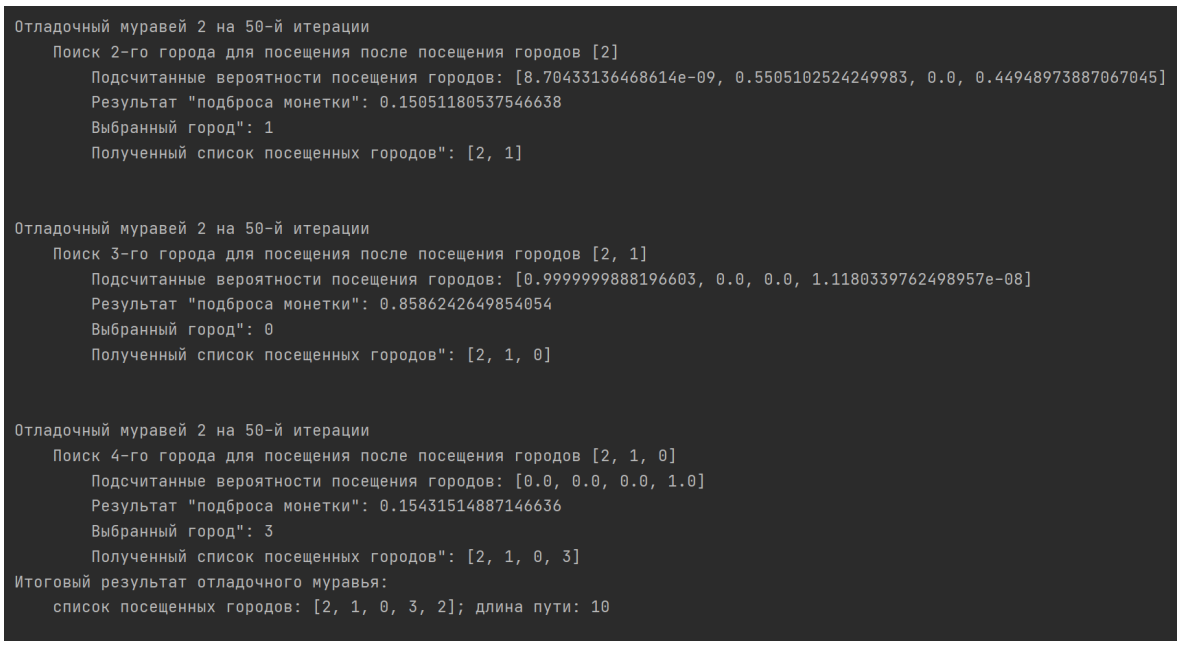

Рисунок 3.3 – Результаты тестирования муравьиного алгоритма, часть 3

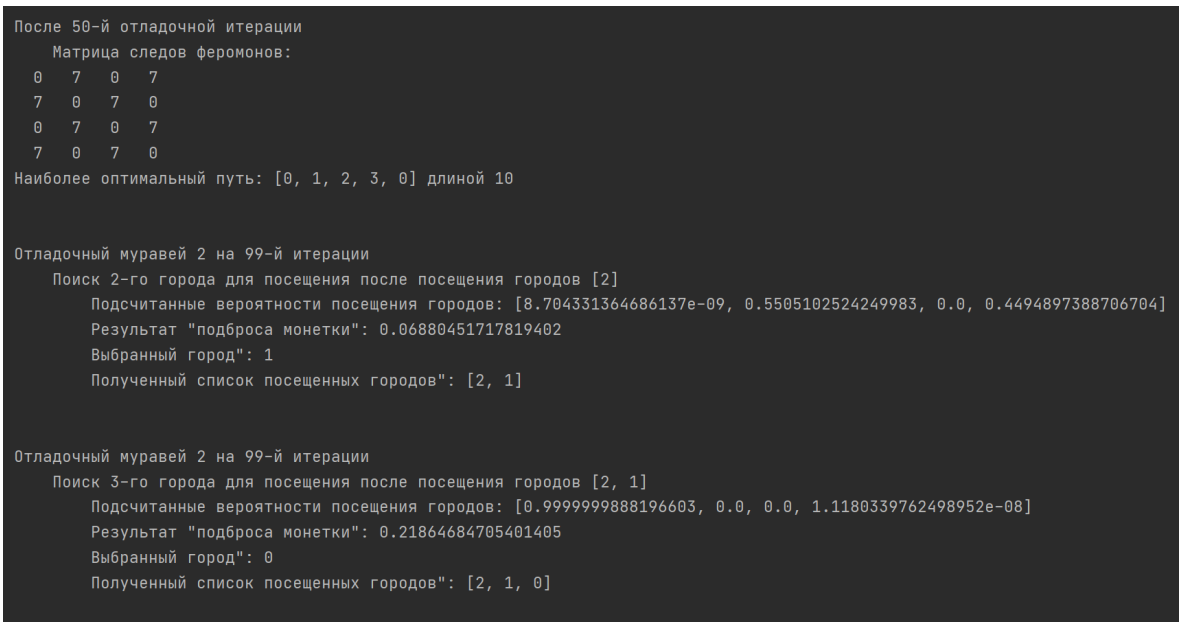

Рисунок 3.4 – Результаты тестирования муравьиного алгоритма, часть 4

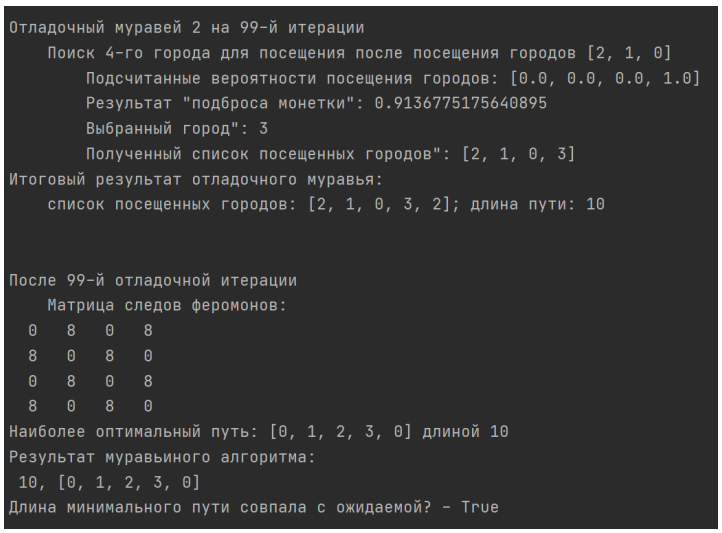

Рисунок 3.5 – Результаты тестирования муравьиного алгоритма, часть 5

Вывод

Был производен выбор средств реализации, приведены требования к ПО, реализованы и протестированы алгоритмы решения задачи коммивояжера.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- операционная система: Windows 10;
- оперативная память: 16 Гб;
- процессор: Intel® Core™ i5-8259U;
- количество ядер: 4;
- количество логических процессоров: 8.

Во время тестирования ноутбук был включен в сеть питания и нагружен только встроенными приложениями окружения и системой тестирования.

4.2 Пример работы программы

На рисунке 4.1 приведен пример работы программы.

```
Введите количество городов: 5
Введите матрицу смежности:
  0 1 3 4 5
  1   0 2 3 4
  3   2   0 4 5
  4   3   4   0 1
  5   4   5   1   0
Введите параметр alpha: 0.9
Введите параметр po: 0.2
Введите параметр tmax: 50
Матрица смежности:
  0   1   3   4   5
  1   0   2   3   4
  3   2   0   4   5
  4   3   4   0   1
  5   4   5   1   0
Алгоритм полного перебора: ответ=13.0, путь=[4, 3, 2, 1, 0, 4]
Муравьиный алгоритм: ответ=13.0, путь=[2, 1, 0, 4, 3, 2]
```

Рисунок 4.1 – Пример работы программы

4.3 Параметризация муравьиного алгоритма

В муравьином алгоритме вычисления проводятся с использованием таких настраиваемых парамтеров, как α (коэффициент жадности), po (коэффициент испарения феромона) и tmax (время жизни колонии). Проведем параметризацию этого алгоритма, то есть подберем такие наборы параметров, при которых алгоритм работает лучше всего, то есть находит решение, наиболее близкое к эталонному. Эталонным будем считать решение, найденное алгоритмом полного перебора.

Чтобы не "подгонять"алгоритм под конкретные входные данные (матрицу смежности), необходимо проводить тестирование на нескольких классах данных. Возьмем три следующих класса: в первом и втором данные генерируются случайным образом, при этом разброс длин путей в первом (small) равен 10, во втором (big) - 1000, в третьем же классе (local) все длины путей задаются равными 12, затем прокладывается единственный

оптимальный маршрут, в котором все города связаны путям длиной 8, а также специально создаются несколько локальных минимумов добавлением нескольких путей длины 10. Чтобы набор параметров претендовал на наилучший, алгоритм с такими настройками должен "хорошо"обработать каждый класс матриц смежности.

Будем рассматривать матрицы размерности 11×11, чтобы получение точного результата муравьиным алгоритмом было более затруднено и сильнее зависело от параметров. Для параметров α и ро будем (независимо друг от друга) задавать значения [0.1, 0.25, 0.5, 0.75, 0.9], а для tmax - [100, 200, 300, 400, 500].

Результаты тестирования приведены на рисунках 4.2-4.6, где $diff_x$ - разница между эталонным решением и решением, полученным с помощью муравьиного алгоритма на классе данных x, а diff_sum - сумма всех разностей, поделенных на разброс длин путей в соответсвующем классе данных. Результаты отсортированы по возрастанию значений колонки diff_sum. Вторым параметром сортировки стал tmax, так как при равном качестве алгоритм, использовавший меньшее число итераций, является более оптимальным.

alpha	po	tmax	diff_big	diff_small	diff_local	diff_sum
0,1	0,75	400	8	3	0	0,308
0,25	0,1	500	161	2	0	0,361
0,25	0,75	300	161	3	0	0,461
0,5	0,75	300	161	3	0	0,461
0,25	0,25	400	161	3	0	0,461
0,25	0,75	500	161	3	0	0,461
0,5	0,9	500	161	3	0	0,461
0,75	0,75	500	161	3	0	0,461
0,5	0,5	200	224	3	0	0,524
0,75	0,5	200	237	3	0	0,537
0,75	0,75	200	237	3	0	0,537
0,25	0,9	300	237	3	0	0,537
0,5	0,25	300	237	3	0	0,537
0,75	0,5	300	237	3	0	0,537
0,5	0,1	400	237	3	0	0,537
0,5	0,25	400	237	3	0	0,537
0,5	0,9	400	237	3	0	0,537
0,75	0,1	400	237	3	0	0,537
0,75	0,25	400	237	3	0	0,537
0,5	0,1	500	237	3	0	0,537
0,5	0,25	500	237	3	0	0,537
0,5	0,5	500	237	3	0	0,537
0,5	0,75	500	237	3	0	0,537
0,75	0,5	500	237	3	0	0,537
0,9	0,1	500	237	3	0	0,537
0,1	0,9	500	149	0	4	0,549
0,5	0,1	300	300	3	0	0,6
0,75	0,1	300	300	3	0	0,6
0,5	0,5	400	300	3	0	0,6
0,75	0,9	400	300	3	0	0,6
0,9	0,75	400	300	3	0	0,6
0,1	0,5	400	8	2	4	0,608

Рисунок 4.2 – Результаты параметризации, часть 1

0,75	0,75	400	318	3	0	0,618
0,75	0,1	200	224	4	0	0,624
0,5	0,5	300	224	4	0	0,624
0,5	0,1	200	237	4	0	0,637
0,25	0,25	200	161	5	0	0,661
0,75	0,25	500	383	3	0	0,683
0,75	0,9	300	300	4	0	0,7
0,9	0,75	500	300	4	0	0,7
0,1	0,25	300	125	2	4	0,725
0,75	0,25	100	237	5	0	0,737
0,75	0,9	200	237	5	0	0,737
0,75	0,25	300	237	5	0	0,737
0,75	0,9	500	237	5	0	0,737
0,9	0,9	500	237	5	0	0,737
0,1	0,75	500	142	2	4	0,742
0,9	0,75	300	450	3	0	0,75
0,9	0,25	200	459	3	0	0,759

Рисунок 4.3 – Результаты параметризации, часть 2

0,9	0,5	400	459	3	0	0,759
0,25	0,75	400	161	2	4	0,761
0,75	0,75	300	396	4	0	0,796
0,75	0,1	500	498	3	0	0,798
0,5	0,9	100	300	5	0	0,8
0,1	0,1	300	0	2	6	0,8
0,1	0,9	300	0	2	6	0,8
0,1	0,1	400	8	4	4	0,808
0,1	0,25	400	8	2	6	0,808
0,9	0,5	200	513	3	0	0,813
0,9	0,5	300	513	3	0	0,813
0,9	0,25	400	513	3	0	0,813
0,9	0,9	400	513	3	0	0,813
0,9	0,9	200	318	5	0	0,818
0,9	0,25	500	318	5	0	0,818
0,25	0,1	400	224	2	4	0,824
0,9	0,25	300	459	4	0	0,859
0,9	0,9	300	459	4	0	0,859
0,25	0,1	200	161	3	4	0,861
0,5	0,75	200	161	3	4	0,861
0,25	0,1	300	161	3	4	0,861
0,25	0,9	400	173	3	4	0,873
0,25	0,5	500	173	3	4	0,873
0,1	0,9	200	0	3	6	0,9
0,1	0,75	100	8	3	6	0,908
0,1	0,5	200	8	3	6	0,908
0,1	0,9	400	8	3	6	0,908
0,25	0,5	400	224	3	4	0,924
0,25	0,1	100	326	2	4	0,926
0,1	0,1	500	32	3	6	0,932

Рисунок 4.4 – Результаты параметризации, часть 3

0,1	0,25	500	32	3	6	0,932
0,25	0,75	200	237	3	4	0,937
0,1	0,5	300	142	2	6	0,942
0,9	0,5	500	644	3	0	0,944
0,5	0,9	200	161	4	4	0,961
0,1	0,9	100	0	2	8	1
0,75	0,5	400	318	3	4	1,018
0,25	0,25	300	224	4	4	1,024
0,1	0,25	200	125	3	6	1,025
0,25	0,25	100	142	3	6	1,042
0,25	0,5	200	142	3	6	1,042
0,1	0,5	500	142	3	6	1,042
0,25	0,25	500	142	3	6	1,042
0,1	0,75	300	154	3	6	1,054
0,9	0,25	100	558	5	0	1,058
0,5	0,5	100	161	3	6	1,061
0,25	0,9	500	161	3	6	1,061
0,25	0,75	100	262	4	4	1,062
0,25	0,5	300	221	3	6	1,121
0,75	0,75	100	628	5	0	1,128

Рисунок 4.5 – Результаты параметризации, часть 4

0,9	0,75	200	730	4	0	1,13
0,9	0,1	400	730	4	0	1,13
0,1	0,1	100	237	3	6	1,137
0,25	0,9	200	237	3	6	1,137
0,5	0,9	300	237	3	6	1,137
0,1	0,75	200	338	2	6	1,138
0,25	0,9	100	161	4	6	1,161
0,9	0,9	100	864	3	0	1,164
0,75	0,25	200	396	4	4	1,196
0,5	0,75	400	300	5	4	1,2
0,1	0,1	200	231	2	8	1,231
0,5	0,25	200	237	4	6	1,237
0,9	0,1	100	541	3	4	1,241
0,9	0,1	300	742	5	0	1,242
0,5	0,1	100	573	3	4	1,273
0,9	0,5	100	781	5	0	1,281
0,1	0,25	100	237	3	8	1,337
0,5	0,25	100	237	5	6	1,337
0,1	0,5	100	142	4	8	1,342
0,75	0,5	100	736	3	4	1,436
0,5	0,75	100	573	5	4	1,473
0,75	0,9	100	573	3	6	1,473
0,9	0,1	200	679	8	0	1,479
0,25	0,5	100	391	5	6	1,491
0,75	0,1	100	300	7	6	1,6
0,9	0,75	100	468	9	6	1,968

Рисунок 4.6 – Результаты параметризации, часть 5

Наилучшим образом на всех классах данных отработал муравьиный алгоритм с параметрами ($\alpha = 0.1$, ро=0.75 и tmax=400), за ним - с параметрами ($\alpha = 0.25$, ро=0.1 и tmax=500) и ($\alpha = 0.25$, ро=0.75 и tmax=300). При проведении таких же экспериментов для матриц меньшей размерности трудно определить 'параметры-победители', так как алгоритм в целом ошибается очень редко.

В целом же по результатам параметризации можно сделать следующие выводы:

1) муравьиный алгоритм хорошо справляется с небольшими графами вне зависимости от параметров;

2) на матрицах достаточно большой размерности (в нашем примере - 11x11) он ошибается чаще и становится более зависимым от параметров;

3) на матрицах большой размерности при ро=const и tmax=const меньшее значение α, которое приближает алгоритм к жадности, приводит

к лучшим результатам в сравнении с большими значениями;

4) при ро=const и α=const большее время жизни колонии tmax позволяет получить ответ более близкий к эталонному.

4.4 Сравнение трудоемкостей реализаций

Задача коммивояжера является NP-трудной, и точный переборный алгоритм ее решения имеет факториальную сложность. Сложность муравьиного алгоритма равна $O(tmax * m * n^2))$, то есть она зависит от времени жизни колонии, количества городов и количества муравьев в колонии. [1]. В данной реализации количество муравьев равно количеству городов, и трудоемкость муравьиного алгоритма равна $O(tmax * n^3)$.

4.5 Сравнение времени выполнения реализаций алгоритмов

Сравнивалось процессорное время работы алгоритма полного перебора и муравьиного алгоритма с параметрами, выбранными в результате параметрзации ($\alpha = 0.1$, ро=0.75 и tmax=400). Эти реализации сравнивались по времени работы при количестве городов от 3 до 11 с шагом 1.

Так как некоторые задачи выполняются достаточно быстро, а замеры времени имеют некоторую погрешность, они для каждой реализации и каждого количества заявок выполнялись 10 раз, а затем вычислялось среднее время работы.

На рисунке 4.7 приведены результаты сравнения времени выполнения реализаций алгоритмов.

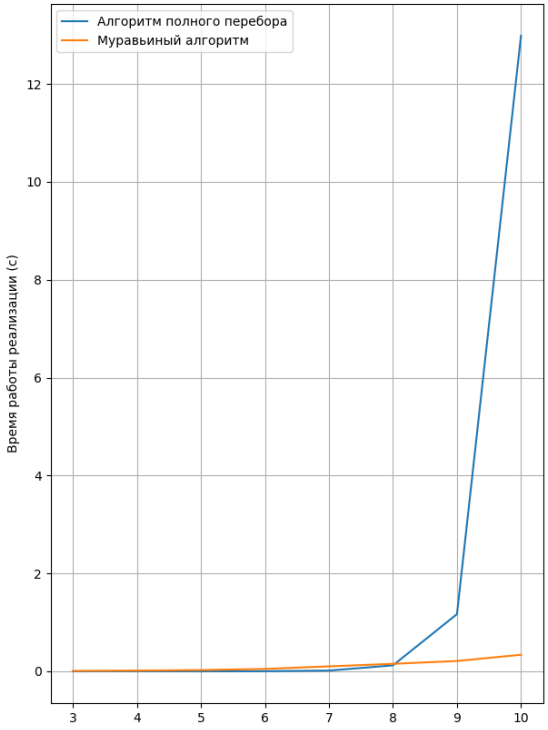

Рисунок 4.7 – Сравнение времени работы реализаций в зависимости от количества городов

Как видно из графиков, теоретическая оценка трудоемкости подтвердилась: время работы алгоритма полного перебора растет как O(n!), а муравьиного алгоритма - как O(n^3), где n - количество городов. В связи с этим при небольшом количестве городов (до 8 включительно) алгоритм полного перебора находит решение за меньшее количество времени по сравнению с муравьиным алгоритмом, однако далее с ростом числа городов начинает все стремительней уступать второму.

4.6 Вывод из исследовательской части

Таким образом, в ситуациях, когда количество городов велико (более 8, например), а задача коммивояжера не требует абсолютно точного ответа, для ее решения можно использовать муравьиный алгоритм вместо алгоритма полного перебора, что позволит сэкономить процессорное время. При этом заранее проведенная параметризация может помочь настро-

ить первый алгоритм так, что и он будет в большинстве случаев выдавать точный ответ.

При небольшом же размере графа (прмерно до размерности 8x8) муравьиный алгоритм с большой вероятностью даст точный ответ, однако в этом случае он не дает выигрыша по времени перед алгортмом полного перебора.

Заключение

В результате выполнения лабораторной работы была достигнута поставленная цель: был реализован муравьиный алгоритм для решения задачи коммивояжера и приобретены навыки параметризации алгоритмов.

В рамках выполнения работы были выаолнены следующие задачи:

1) реализован алгоритм полного перебора для решения задачи коммивояжера;

2) изучен и реализован муравьиный алгоритм для решения задачи коммивояжера;

3) проведена параметризаця муравьиного алгоритма на трех классах данных и подобраны оптимальные параметры;

4) проведен сравнительный анализ времени выполнения и трудоемкостей реализаций.

Литература

[1] Ульянов М. В. РЕСУРСНО-ЭФФЕКТИВНЫЕ КОМПЬЮТЕРНЫЕ АЛГОРИТМЫ. РАЗРАБОТКА И АНАЛИЗ // НАУКА ФИЗМАТЛИТ. 2007. С. 201–205.

[2] Задача коммивояжера [Электронный ресурс]. Режим доступа: http://www.math.nsc.ru/LBRT/k5/OR-MMF/TSPr.pdf (дата обращения: 28.10.2021).

[3] Алгоритмы решения задачи коммивояжера [Электронный ресурс]. Режим доступа: https://scienceforum.ru/2021/article/2018025171 (дата обращения: 28.10.2021).

[4] Лутц Марк. Изучаем Python, том 1, 5-е изд. Пер. с англ. — СПб.: ООО “Диалектика”, 2019. Т. 832.

[5] time — Time access and conversions [Электронный ресурс]. Режим доступа: https://docs.python.org/3/library/time.html (дата обращения: 05.09.2021).

[6] Python и Pycharm [Электронный ресурс]. Режим доступа: https://py-charm.blogspot.com/2017/09/pycharm.html (дата обращения: 05.09.2021).