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»

НАПРАВЛЕНИЕ ПОДГОТОВКИ «09.03.04 Программная инженерия»

ОТЧЕТ

ПО ЛАБОРАТОРНОЙ РАБОТЕ №6

Название: Муравьиный алгоритм для решения задачи коммивояжёр

Дисциплина: Анализ алгоритмов

Студент	ИУ7-54Б		С. Д. Параскун
	Группа	Подпись, дата	И. О. Фамилия
Преподаватель			Л. Л. Волкова
		Подпись, дата	И. О. Фамилия

Москва, 2021 г.

Содержание

Страница

Введение

Одна из самых известных и важных задач транспортной логистики (и комбинаторной оптимизации) – задача коммивояжера или «задача о странствующем торговце» (англ. «Travelling Salesman Problem», TSP). [1] Суть задачи сводится к поиску оптимального (кратчайшего, быстрейшего или самого дешевого) пути, проходящего через промежуточный пункты по одному разу и возвращающегося в исходную точку. К примеру, нахождение наиболее выгодного маршрута, позволяющего коммивояжеру посетить со своим товаром определенные города по одному разу и вернуться обратно. Мерой выгодности маршрута может быть минимальное время поездки, минимальные расходы на дорогу или минимальная длина пути. В наше время, когда стоимость доставки часто бывает сопоставима со стоимостью самого товара, а скорость доставки — один из главных приоритетов, задача нахождения оптимального маршрута приобретает огромное значение.

Муравьиные алгоритмы [2] 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, не смотря 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.

Целью данной лабораторной работы является является изучение муравьиного алгоритма. Для достижения поставленной цели необходимо выполнить следующие задачи:

- изучить понятие муравьиного алгоритма на примере решения задачи коммивояжёра;
- изучить решение этой задачи с помощью метода полного перебора;
- составить схемы алгоритмов;
- реализовать разработанные алгоритмы;

- провести параметризацию муравьиного алгоритма для выбранных классов задач;
- провести сравнительный анализ скорости работы реализованных алгоритмов;
- описать и обосновать полученные результаты.

1. Аналитический раздел

В данном разделе будут представлены теоретические сведения о рассматриваемых алгоритмах.

1.1 Полный перебор

Пронумеруем все города от 1 до n. Базовому городу присвоим номер n. Каждый тур по городам однозначно соответствует перестановке целых чисел от 1 до n - 1.

Задачу коммивояжера можно решить образуя все перестановки первых n - 1 целых положительных чисел. Для каждой перестановки строится соответствующий тур и вычисляется его стоимость. Обрабатывая таким образом все перестановки, запоминается тур, который к текущему моменту имеет наименьшую стоимость. Если находится тур с более низкой стоимостью, то дальнейшие сравнения производятся с ним. Сложность алгоритма полного перебора составляет O(n!). [3]

1.2 Муравьиный алгоритм

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, следовательно, большее количество муравьев будет включать его в синтез собственных маршрутов. Моделирование такого подхода, использующего только положительную обратную связь, приводит к преждевременной сходимости — большинство муравьев двигается по локально оптимальному маршруту. Избежать, этого можно, моделируя отрицательную обратную связь в виде испарения феромона. При этом если феромон испаряется быстро, то это приводит к потере па-

мяти колонии и забыванию хороших решений, с другой стороны, большое время испарения может привести к получению устойчивого локально оптимального решения. Теперь, с учетом особенностей задачи коммивояжера, мы можем описать локальные правила поведения муравьев при выборе пути.

- Муравьи имеют собственную «память». Поскольку каждый город может быть посещен только один раз, у каждого муравья есть список уже посещенных городов — список запретов. Обозначим через J_{ik} 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};

$$\eta_{ij} = 1/D_{ij} \tag{1.1}$$

- 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 $\tau_{ij}(t)$.

Исходя из этого можем сформировать правило, по которому k-ый муравей может перейти из города i в город j:

$$P_{kij} = \begin{cases} \frac{\tau_{ij}^a \eta_{ij}^b}{\sum_{q=1}^{m} \tau_{iq}^a \eta_{iq}^b}, j \in J_{ik} \\ 0, \text{иначе} \end{cases} \tag{1.2}$$

где a, b — параметры, задающие веса следа феромона, при a = 0 алгоритм вырождается до жадного алгоритма (будет выбран ближайший город). Заметим, что выбор города является вероятностным, правило 2 лишь определяет ширину зоны города j; в общую зону всех городов J_{ik} ; бросается случайное число, которое и определяет выбор муравья.

Правило 2 не изменяется в ходе алгоритма, но у двух разных муравьев значение вероятности перехода будут отличаться, т. к. они имеют разный

список разрешенных городов. Пройдя ребро (i, j), муравей откладывает на нем некоторое количество феромона, которое должно быть связано с оптимальностью сделанного выбора. 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:

$$P_{kij} = \begin{cases} Q/L_k(t), (i,j) \in T_k(t) \\ 0, \text{иначе} \end{cases} \tag{1.3}$$

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

$$\tau_{ij}(t) = (1-\rho) \cdot \tau_{ij}(t) + \Delta\tau_{ij}(t); \Delta\tau_{ij}(t) = \sum_{k=1}^{m} \Delta\tau_{ijk}(t) \tag{1.4}$$

где m - количество муравьев в колонии.

В начале алгоритма количество феромона на ребрах принимается равным небольшому положительному числу. Общее количество муравьев остается постоянным и равным количеству городов, каждый муравей начинает маршрут из своего города. Сложность алгоритма: $O(t_{max} \cdot max(m, n^2))$, где t_{max} — время жизни колонии, m — количество муравьев в колонии, n — размер графа.

1.3 Вывод

В данном разделе были рассмотрены алгоритмы поиска кратчайшего расстояния между городами. Полученных знаний достаточно для разработки. На вход алгоритмам будет подаваться количество городов n и матрица расстояний между городами размера $n \cdot n$. Затем вводятся коэффициенты a от 0 до 1, коэффициент испарения p от 0 до 1, количество итераций t и количество "элитных"муравьев e.

Реализуемое ПО будет работать в пользовательском режиме (вывод результатов), а также в экспериментальном (проведение замеров времени выполнения алгоритмов).

2. Конструкторский раздел

В данном разделе будут спроектированы схемы алгоритмов, описаны используемые типы данных, а также произведена оценка памяти и описана структура ПО.

2.1 Схемы алгоритмов

На рисунках 2.1 - 2.2 представлены схемы рассматриваемых алгоритмов.

На рисунках 2.3 - 2.5 представлены схемы используемых в муравьином алгоритме функций.

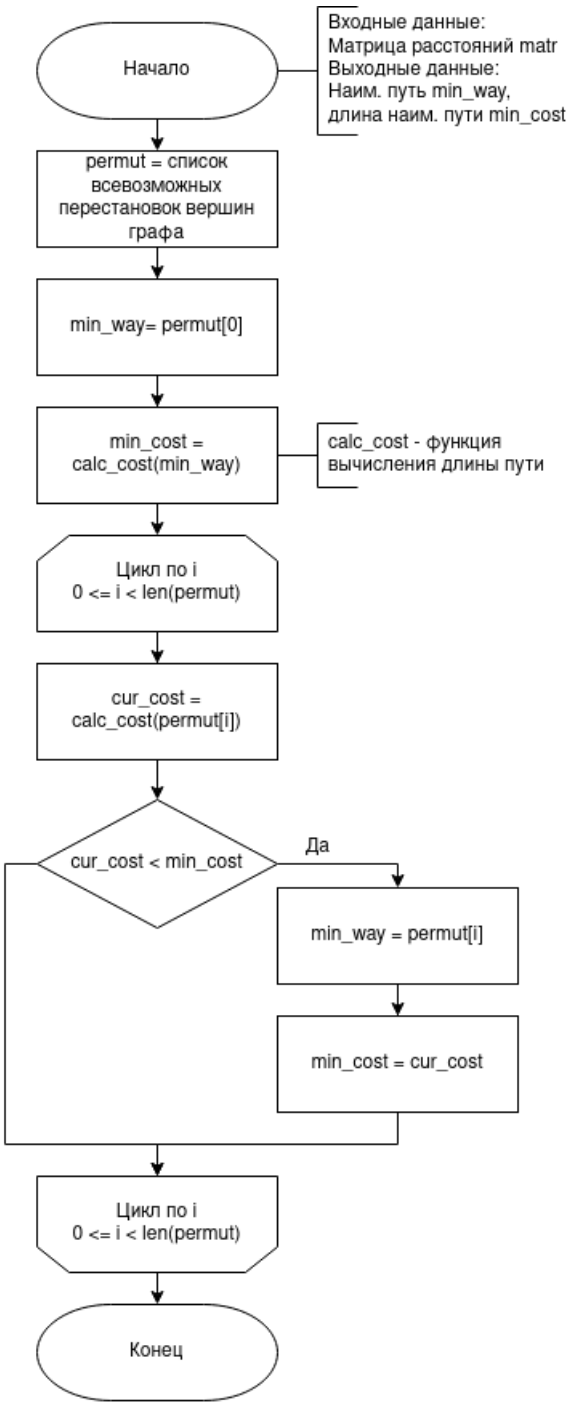

Рисунок 2.1 – Схема алгоритма полного перебора

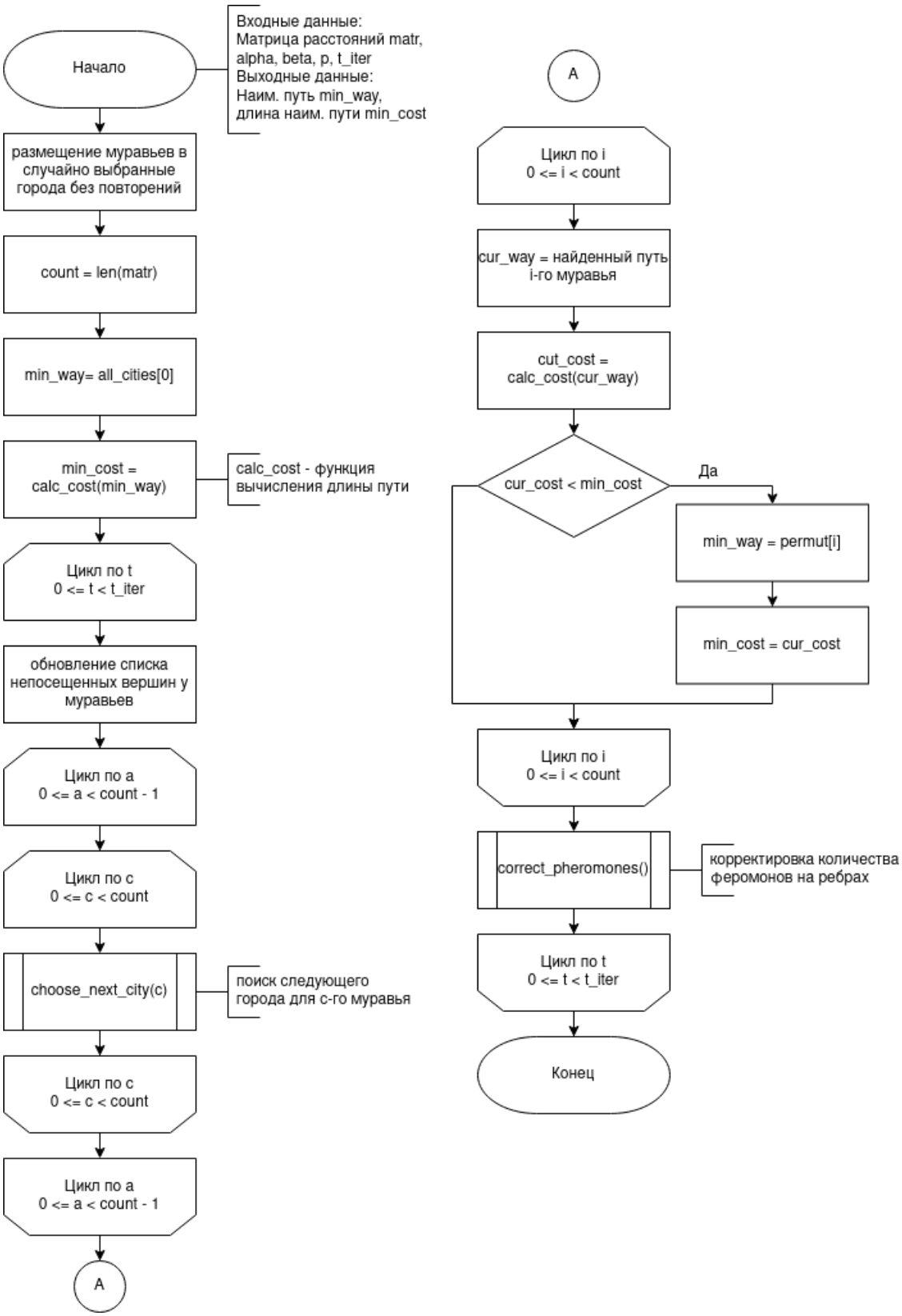

Рисунок 2.2 – Схема муравьиного алгоритма

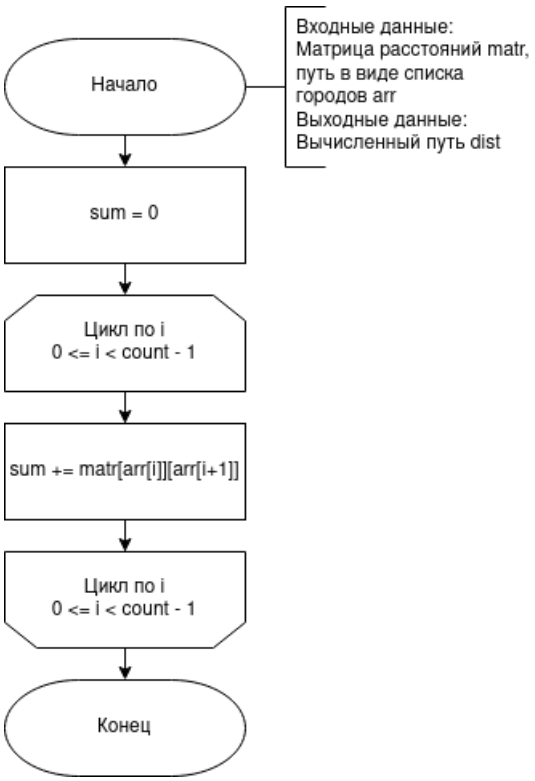

Рисунок 2.3 – Функция вычисления расстояния пути

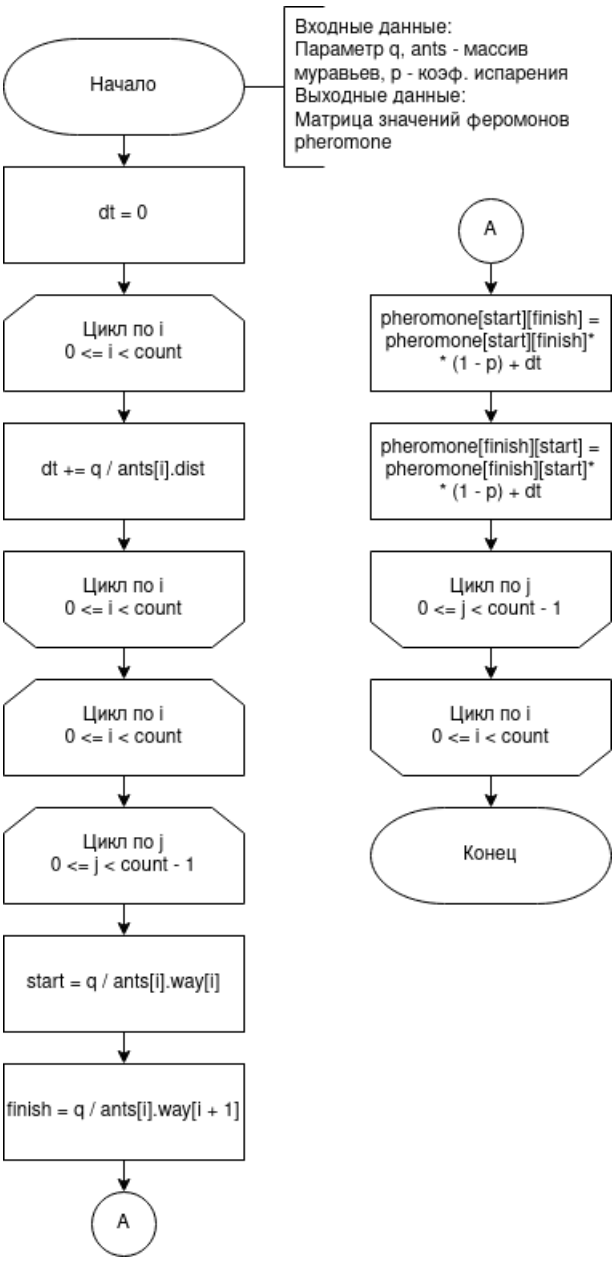

Рисунок 2.4 – Функция корректировки феромонов

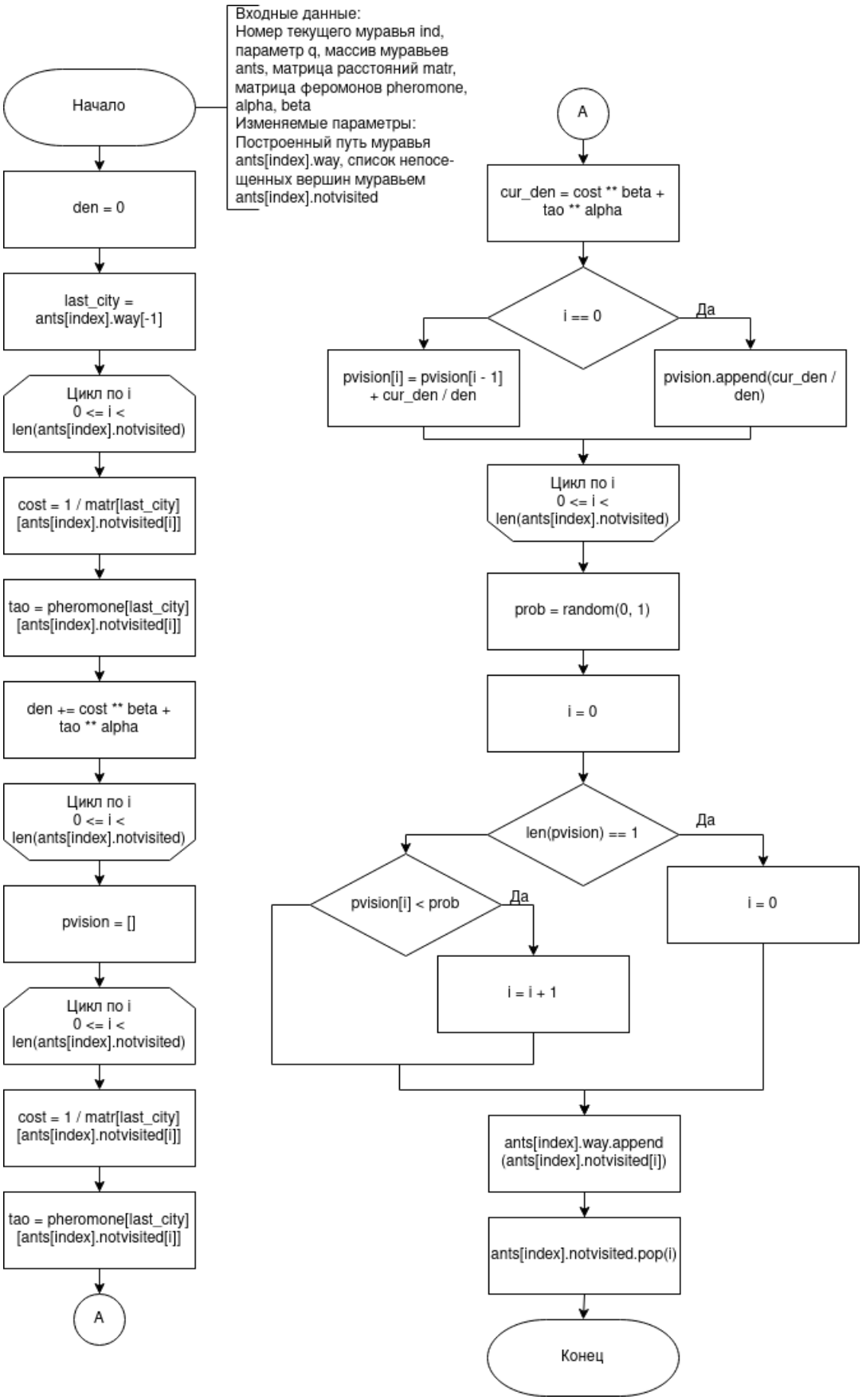

Рисунок 2.5 – Функция выбора муравьем следующего города

2.2 Используемые типы данных

При реализации алгоритмов будут использованы следующие структуры данных:

- муравьиная колония AntColony;

Листинг 2.1 – Класс AntColony

```
class AntColony(object):
    def __init__(self, distances, decay, tmax, alpha=0.5,\
     beta=0.5, e=0):
        self.distances = distances
        for i in range(len(distances)):
            for j in range(len(distances)):
                if self.distances[i][j] == 0:
                    self.distances[i][j] = 10000000
        self.pheromone = np.ones(self.distances.shape)\
         / len(distances)
        self.all_inds = list(range(len(distances)))
        self.decay = decay
        self.alpha = alpha
        self.beta = beta
        self.tmax = tmax
        self.count = len(distances)
        self.ants = []
        self.n_elits = e
        self.q = sum([sum(x) for x in distances])*2
```

Она имеет следующие поля:

1. distances - матрица расстояний;
2. pheromone - матрица значений феромонов;
3. all_inds - упорядоченный список всех вершин графа;
4. decay - коэффициент испарения;
5. alpha - коэффициент α;
6. beta - коэффициент β;
7. tmax - количество итераций;
8. count - количество вершин в графе;
9. ants - список объектов муравьёв;

10. n_elits - количество "элитных"муравьёв;
11. q - коэффициент q.

- муравей Ant.

Листинг 2.2 – Класс Ant

```
class Ant(object):
    def __init__(self):
        self.way = []
        self.start = 0
        self.dist = 0
        self.notvisited = []
```

Она имеет следующие поля:

1. way - текущий путь муравья;
2. start - вершина, с которой муравей начинает свой путь;
3. dist - длина найденного кратчайшего пути;
4. notvisited - список непосещенных вершин.

2.3 Оценка памяти

Рассмотрим затрачиваемый объем памяти для рассмотренных алгоритмов.

При полном переборе память используется на:

- размер матрицы расстояний n - sizeof(int);
- матрица расстояний - n * n * sizeof(int);
- вспомогательные переменные - 4 * sizeof(int);
- вспомогательный список всевозможных перестановок вершин графа - n! * sizeof(int).

При использовании муравьиного алгоритма помимо матрицы расстояний и ее размера используются также:

- размер полей класса AntColony - sizeof(AntColony);

- муравьи класса Ant - k * sizeof(Ant);
- вспомогательные переменные - 15 * sizeof(double).

Таким образом, муравьиный алгоритм значительно увеличивает количество используемой памяти из-за используемых структур и классов, а также большего числа вспомогательных переменных.

2.4 Структура ПО

ПО будет состоять из следующих модулей:

- главный модуль - из него будет осуществляться запуск программы и выбор соответствующего режима работы;
- модуль интерфейса - в нем будет описана реализация режимов работы программы;
- модуль, содержащий реализации алгоритмов.

2.5 Вывод

На основе полученных в аналитическом разделе знаний об алгоритмах были спроектированы схемы алгоритмов, выбраны используемые типы данных, проведена оценка затрачиваемого объема памяти, а также описана структура ПО.

3. Технологический раздел

В данном разделе будут приведены требования к ПО, средства его реализации и листинга кода алгоритмов, а также рассмотрены тестовые случаи.

3.1 Требования к ПО

Программное обеспечение должно удовлетворять следующим требованиям:

- ПО принимает матрицу расстояний и требуемые коэффициенты;
- ПО возвращает кратчайший путь и его длину.

3.2 Средства реализации

Для реализации ПО был выбран язык python. В качестве среды разработки - VisualStudio Code. Оба средства были выбраны из тех соображений, что навыки работы с ними были получены в более ранних курсах.

3.3 Листинги кода

Листинги 3.1 - 3.6 демонстрируют реализацию алгоритмов.

Листинг 3.1 – Функция вычисляющая длину пути

```
def calc_dist(array, distances):
    cost = 0
    for i in range(len(array) - 1):
    cost += distances[array[i]][array[i + 1]]
    return cost
```

Листинг 3.2 – Алгоритм полного перебора

```
def permut(matr):
    arr = list(range(len(matr)))
    permut_arr = list(itertools.permutations(arr))
```

```
    for i in range(len(permut_arr)):
        permut_arr[i] = list(permut_arr[i])
        permut_arr[i].append(permut_arr[i][0])
    costs = [calc_dist(x, matr) for x in permut_arr]
    return min(costs), permut_arr[costs.index(min(costs))]
```

Листинг 3.3 – Муравьиный алгоритм

```
def run(self):
    self.ants = self.allocation_ants()
    best_way = copy.deepcopy(self.all_inds)
    best_way.append(self.all_inds[0])
    least_dist = self.calc_dist(best_way)
    for t in range(int(self.tmax)):
        self.ants = self.allocation_ants()
        for a in range(self.count - 1):
            self.choose_way()
        for i in range(self.count):
            self.ants[i].way.append(self.ants[i].way[0])
            self.ants[i].dist = self.calc_dist(self.ants[i].way)
            if self.ants[i].dist < least_dist:
                least_dist = self.ants[i].dist
                best_way = copy.deepcopy(self.ants[i].way)
        dt = self.correct_pheromones()
        self.elit(best_way, dt)
    return best_way
```

Листинг 3.4 – Функция корректировки ферамонов

```
def correct_pheromones(self):
    delta_tao = 0
    for i in range(self.count):
        delta_tao += self.q / self.ants[i].dist
    for i in range(self.count):
        for j in range(len(self.ants[i].way)-1):
            start = self.ants[i].way[j]
            finish = self.ants[i].way[j+1]
            self.pheromone[start][finish] = self.pheromone[start][finish]\
             * (1 - self.decay) + delta_tao
            self.pheromone[finish][start] = self.pheromone[finish][start]\
             * (1 - self.decay) + delta_tao
            if self.pheromone[start][finish] < 1 / len(self.distances):
                self.pheromone[start][finish] = 1 / len(self.distances)
            if self.pheromone[finish][start] < 1 / len(self.distances):
                self.pheromone[finish][start] = 1 / len(self.distances)
    return delta_tao
```

Листинг 3.5 – Функция распределения муравьёв по случайным городам

```
def allocation_ants(self):
    starts = []
    ants = []
    for i in range(self.count):
        cur_ant = ant()
        a = randint(0, self.count - 1)
        while a in starts:
            a = randint(0, self.count - 1)
        starts.append(a)
        cur_ant.start = a
        cur_ant.way.append(a)
        cur_ant.notvisited = copy.deepcopy(self.all_inds)
        cur_ant.notvisited.remove(a)
        ants.append(cur_ant)
    return ants
```

Листинг 3.6 – Функции выбора следующего города для муравьёв

```
def choose_city(self, index):
    denominator = 0
    last_city = self.ants[index].way[-1]
    for i in range(len(self.ants[index].notvisited)):
        cost = 1 / self.distances[last_city][self.ants[index].notvisited[i]]
        tao = self.pheromone[last_city][self.ants[index].notvisited[i]]
        denominator += cost ** self.beta + tao ** self.alpha
    pvision = []
    for i in range(len(self.ants[index].notvisited)):
        cost = 1 / self.distances[last_city][self.ants[index].notvisited[i]]
        tao = self.pheromone[last_city][self.ants[index].notvisited[i]]
        p = cost ** self.beta + tao ** self.alpha
        if i == 0:
            pvision.append(p/denominator)
        else:
            pvision.append(pvision[i - 1] + p/denominator)
    probability = random.random()
    i = 0
    if len(pvision) == 1:
        i = 0
    else:
        while pvision[i] < probability:
            i += 1
    self.ants[index].way.append(self.ants[index].notvisited[i])
    self.ants[index].notvisited.pop(i)
def choose_way(self):
    for i in range(self.count):
        self.choose_city(i)
```

3.4 Тестирование ПО

В таблице 3.1 описаны ожидаемый и действительный результат для рассматриваемых алгоритмов. Эти два результата совпали.

Таблица 3.1 – Тестирование функций

Матрица смежности	Ожидаемый результат	Действительный результат
$\begin{pmatrix} 0 & 1 & 10 & 7 \\ 1 & 0 & 1 & 2 \\ 10 & 1 & 0 & 1 \\ 7 & 2 & 1 & 0 \end{pmatrix}$	10, [0, 1, 2, 3, 0]	10, [0, 1, 2, 3, 0]
$\begin{pmatrix} 0 & 3 & 5 & 7 \\ 7 & 0 & 1 & 2 \\ 5 & 1 & 0 & 1 \\ 7 & 2 & 1 & 0 \end{pmatrix}$	11, [2, 3, 1, 0, 2]	11, [2, 3, 1, 0, 2]
$\begin{pmatrix} 0 & 3 & 5 \\ 3 & 0 & 1 \\ 5 & 1 & 0 \end{pmatrix}$	9, [0, 1, 2, 0]	9, [0, 1, 2, 0]

3.5 Вывод

На основе схем из конструкторского раздела были написаны реализации требуемых алгоритмов, а также проведено их тестирвание.

4. Исследовательский раздел

В данном разделе будет проведен сравнительный анализ алгоритмов по времени и затрачиваемой памяти.

4.1 Технические характеристики

Тестирование выполнялось на устройстве со следующими характеристиками:

- Операционная система Ubuntu 20.04.3 LTS [4]
- Память 16 Гб
- Процессор Intel Core i5-1135G7 11th Gen, 2.40 Гц [5]

Во время проведения эксперимента устройство не было нагружено сторонними задачами, а также было подключено к блоку питания.

Процессорное время работы алгоритмов было замерено с помощью функции process_time библиотеки time . Эта функция возвращает значение в долях секунды суммы системного и пользовательского процессорного времени текущего процесса, не включая время, прошедшее во время сна. Контрольная точка возвращаемого значения не определена, поэтому допустима только разница между результатами последовательных вызовов. [6]

4.2 Оценка времени работы алгоритмов

Замеры для каждого размера проводились 10 раз. В качестве результата взято среднее время работы алгоритма при данном размере матриц. Матрицы заполнялись случайными целыми числами в диапазоне от 0 до 10 000. Муравьиный алгоритм выполнялся при следующих параметрах: $\alpha = 0.5$, p = 0.2, t_iters = 8000, elits = count / 3. Результаты замеров приведены в таблице 4.1.

Таблица 4.1 – Таблица времени выполнения реализаций

Кол-во вершин графа	Время полн. перебора(с)	Время муравьиного алг.(с)
3	1.7	4.3e-05
5	5.0	0.00062
7	1.1e+01	0.036
9	2.2e+01	3.1
11	37	440

Можно сделать вывод, что для графов, размером до 10 вершин алгоритм полного перебора оказывается выгоднее муравьиного алгоритма, однако при n = 11, второй алгоритм работает быстрее первого в 11.9 раз. С увеличением количества вершин в графе это преимущество будет возрастать, так как время работы алгоритма полного перебора будет увеличиваться быстрее за счёт сложности алгоритма O(n!).

4.3 Параметризация муравьиного алгоритма

Для каждого набора α, β, p найдём минимальное количество итераций t_iter , для которого результат будет наиболее точным. Если за t_iter = 10000 добиться правильного результата (заранее вычисленного с помощью алгоритма полного перебора) не удаётся, рассчитаем, на сколько полученный результат отличается от верного.

Подобный эксперимент проводится для трёх классов задач:

1. многие вершины в графе не связаны между собой;
2. расстояния в матрице расстояний незначительно отличаются друг от друга;
3. расстояния в матрице значительно отличаются друг от друга.

Полученные результаты представлены на таблицах 4.2 - 4.4.

Таблица 4.2 – Таблица параметризации для первого класса задач

α	β	p	t_{iter}	dif
0.0	1.0	0	11500	317.0
0.0	1.0	0.3	8500	0
0.0	1.0	0.5	9000	0
0.0	1.0	0.7	11500	317.0
0.0	1.0	1	6500	0
0.1	0.9	0	7000	0
0.1	0.9	0.3	9000	0
0.1	0.9	0.5	4500	0
0.1	0.9	0.7	6000	0
0.1	0.9	1	6500	0
0.2	0.8	0	5000	0
0.2	0.8	0.3	5500	0
0.2	0.8	0.5	6000	0
0.2	0.8	0.7	6000	0
0.2	0.8	1	6000	0
0.3	0.7	0	6000	0
0.3	0.7	0.3	6500	0
0.3	0.7	0.5	4500	0
0.3	0.7	0.7	4500	0
0.3	0.7	1	5000	0
0.4	0.6	0	5500	0
0.4	0.6	0.3	6000	0
0.4	0.6	0.5	5000	0
0.4	0.6	0.7	4000	0
0.4	0.6	1	5000	0
0.5	0.5	0	2000	0
0.5	0.5	0.3	5500	0
0.5	0.5	0.5	3000	0
0.5	0.5	0.7	3000	0
0.5	0.5	1	2000	0

α	β	p	t_{iter}	dif
0.6	0.4	0	1500	0
0.6	0.4	0.3	5000	0
0.6	0.4	0.5	500	0
0.6	0.4	0.7	2500	0
0.6	0.4	1	500	0
0.7	0.3	0	1500	0
0.7	0.3	0.3	2500	0
0.7	0.3	0.5	5000	0
0.7	0.3	0.7	1500	0
0.7	0.3	1	1500	0
0.8	0.2	0	500	0
0.8	0.2	0.3	3500	0
0.8	0.2	0.5	500	0
0.8	0.2	0.7	500	0
0.8	0.2	1	1000	0
0.9	0.1	0	1000	0
0.9	0.1	0.3	1500	0
0.9	0.1	0.5	1000	0
0.9	0.1	0.7	1000	0
0.9	0.1	1	1000	0
1.0	0.0	0	500	0
1.0	0.0	0.3	1000	0
1.0	0.0	0.5	1000	0
1.0	0.0	0.7	1000	0
1.0	0.0	1	500	0

Таблица 4.4 – Таблица параметризации для второго класса задач

α	β	p	t_{iter}	dif
0.0	1.0	0	8000	0
0.0	1.0	0.3	6000	0
0.0	1.0	0.5	5000	0
0.0	1.0	0.7	7000	0
0.0	1.0	1	5500	0
0.1	0.9	0	6000	0
0.1	0.9	0.3	6000	0
0.1	0.9	0.5	7000	0
0.1	0.9	0.7	5000	0
0.1	0.9	1	6000	0
0.2	0.8	0	6000	0
0.2	0.8	0.3	6000	0
0.2	0.8	0.5	5500	0
0.2	0.8	0.7	5000	0
0.2	0.8	1	6000	0
0.3	0.7	0	6000	0
0.3	0.7	0.3	5500	0
0.3	0.7	0.5	5000	0
0.3	0.7	0.7	6000	0
0.3	0.7	1	3000	0
0.4	0.6	0	6000	0
0.4	0.6	0.3	3500	0
0.4	0.6	0.5	5000	0
0.4	0.6	0.7	5500	0
0.4	0.6	1	4000	0
0.5	0.5	0	4000	0
0.5	0.5	0.3	6000	0
0.5	0.5	0.5	3500	0
0.5	0.5	0.7	3500	0
0.5	0.5	1	6000	0

α	β	p	t_{iter}	dif
0.6	0.4	0	1000	0
0.6	0.4	0.3	5000	0
0.6	0.4	0.5	3500	0
0.6	0.4	0.7	2500	0
0.6	0.4	1	1000	0
0.7	0.3	0	1000	0
0.7	0.3	0.3	4500	0
0.7	0.3	0.5	2000	0
0.7	0.3	0.7	1500	0
0.7	0.3	1	500	0
0.8	0.2	0	500	0
0.8	0.2	0.3	3000	0
0.8	0.2	0.5	1500	0
0.8	0.2	0.7	1000	0
0.8	0.2	1	1000	0
0.9	0.1	0	1000	0
0.9	0.1	0.3	2000	0
0.9	0.1	0.5	1000	0
0.9	0.1	0.7	500	0
0.9	0.1	1	500	0
1.0	0.0	0	500	0
1.0	0.0	0.3	4000	0
1.0	0.0	0.5	500	0
1.0	0.0	0.7	500	0
1.0	0.0	1	500	0

Таблица 4.6 – Таблица параметризации для третьего класса задач

α	β	p	t_{iter}	dif
0.0	1.0	0	9000	0
0.0	1.0	0.3	10500	66875.0
0.0	1.0	0.5	9500	0
0.0	1.0	0.7	10500	66875.0
0.0	1.0	1	9000	0
0.1	0.9	0	8000	0
0.1	0.9	0.3	5500	0
0.1	0.9	0.5	5500	0
0.1	0.9	0.7	5500	0
0.1	0.9	1	6500	0
0.2	0.8	0	4500	0
0.2	0.8	0.3	6000	0
0.2	0.8	0.5	7500	0
0.2	0.8	0.7	6000	0
0.2	0.8	1	5000	0
0.3	0.7	0	6000	0
0.3	0.7	0.3	7000	0
0.3	0.7	0.5	4000	0
0.3	0.7	0.7	6500	0
0.3	0.7	1	5500	0
0.4	0.6	0	5500	0
0.4	0.6	0.3	5000	0
0.4	0.6	0.5	6500	0
0.4	0.6	0.7	5500	0
0.4	0.6	1	5000	0
0.5	0.5	0	2000	0
0.5	0.5	0.3	4500	0
0.5	0.5	0.5	2500	0
0.5	0.5	0.7	5000	0
0.5	0.5	1	2000	0
0.6	0.4	0	4000	0
0.6	0.4	0.3	3000	0
0.6	0.4	0.5	2500	0
0.6	0.4	0.7	3500	0
0.6	0.4	1	2500	0
0.7	0.3	0	1500	0
0.7	0.3	0.3	4000	0
0.7	0.3	0.5	2000	0
0.7	0.3	0.7	4500	0
0.7	0.3	1	500	0
0.8	0.2	0	1000	0
0.8	0.2	0.3	6000	0
0.8	0.2	0.5	2500	0
0.8	0.2	0.7	1500	0
0.8	0.2	1	1500	0
0.9	0.1	0	500	0
0.9	0.1	0.3	2500	0
0.9	0.1	0.5	3000	0
0.9	0.1	0.7	2000	0
0.9	0.1	1	500	0
1.0	0.0	0	1000	0
1.0	0.0	0.3	3500	0
1.0	0.0	0.5	3000	0
1.0	0.0	0.7	3000	0
1.0	0.0	1	2500	0

4.4 Вывод

Для матрицы расстояний, незначительно отличающихся друг от друга, можно сделать следующие выводы:

1. максимальное число итераций, при котором достигается верный ре-

зультат, равен 8000 для $\alpha = 0$, $\beta = 1$, p = 0;

2. наилучшие результаты достигаются при $\alpha >= 0.9$, p > 0. При таких значениях параметров верный результат достигается за 500 итераций;

3. в среднем значения количества итераций лучше для $\alpha >= 0.6$.

Для матрицы расстояний, значительно отличающихся друг от друга, можно сделать следующие выводы:

1. максимальное число итераций, при котором достигается верный результат, равен 9500 для $\alpha = 0$, $\beta = 1$, p = 0.5;

2. для $\alpha = 0$, $\beta = 1$, p = 0.3 и $\alpha = 0$, $\beta = 1$, p = 0.7 за максимальное количество итераций 10000 не удалось достигнуть верного результата и полученное значение значительно отличается от него;

3. наилучшие результаты достигаются при $\alpha = 0.7$, p = 1. При таких значениях параметров верный результат достигается за 500 итераций;

4. лля каждого набора α, β наилучший результат будет достигаться при наибольшем p. Это показывает, что для матриц расстояний, в которых расстояния сильно отличаются друг от друга, коэффициент испарения должен быть максимальным, чтобы усилить влияние длины пути на выбор пути.

Заключение

В ходе выполнения данной лабораторной работы были выполнены следующие задачи:

- изучено понятие муравьиного алгоритма на примере решения задачи коммивояжёра;
- изучено решение этой задачи с помощью метода полного перебора;
- составлены схемы данных алгоритмов;
- реализованны разработанные версии алгоритмов;
- проведена параметризацию муравьиного алгоритма для выбранных классов задач;
- проведен сравнительный анализ скорости работы реализованных алгоритмов;
- описаны и обоснованы полученные результаты.

Экспериментально были установлены различия в производительности муравьиного алгоритма и метода полного перебора. Для графов с небольшим количеством вершин выгоднее использовать второй алгоритм, однако с увеличением числа вершин в графе, время работы реализации полного перебора значительно возрастает, поэтому для графов, в которых количество вершин > 10, выгоднее использовать муравьиный алгоритм. Уже при n $= 11$ данный алгоритм оказывается эффективнее в ≈ 12 раз. Выбор параметров для муравьиного алгоритма зависит от значений матрицы смежности и от конкретного условия задачи.

Список литературы

[1] В. Левитин А. Алгоритмы. Введение в разработку и анализ. – М.: Вильямс, 2006. Т. 3 из *Глава 3. Метод грубой силы: Задача коммивояжёра.* С. 159–160.

[2] Д. Штовба С. Муравьиные алгоритмы. Exponenta Pro., 2004.

[3] Алгоритмы полного перебора [Электронный ресурс]. Режим доступа: `http://edu.mmcs.sfedu.ru/pluginfile.php/32742/mod_resource/content/\T2A\CYRP\T2A\cyre\T2A\cyrr\T2A\cyre\T2A\cyrb\T2A\cyro\T2A\cyrr\T2A\cyrn\T2A\cyrery\T2A\cyre_\T2A\cyra\T2A\cyrl\T2A\cyrg\T2A\cyro\T2A\cyrr\T2A\cyri\T2A\cyrt\T2A\cyrm\T2A\cyrery.pdf` (дата обращения: 14.12.2021).

[4] Ubuntu 20.04 LTS [Электронный ресурс]. Режим доступа: `https://releases.ubuntu.com/20.04/` (дата обращения: 27.11.2021).

[5] Процессор Intel Core i5-1135G7 [Электронный ресурс]. Режим доступа: `https://www.intel.ru/content/www/ru/ru/products/sku/208658/intel-core-i51135g7-processor-8m-cache-up-to-4-20-ghz/specifications.html` (дата обращения: 27.11.2021).

[6] Time access and conversions [Электронный ресурс]. Режим доступа: `https://docs.python.org/3/library/time.html` (дата обращения: 14.12.2021).