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»

Лабораторная работа №6
по дисциплине «Анализ Алгоритмов»

Тема Методы решения задачи коммивояжера

Студент Куликов Е. А.

Группа ИУ7-56Б

Преподаватель Волкова Л. Л.

Москва — 2024 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В данной рабораторной работе изучаются алгоритмы решения задачи коммивояжера. Задача коммивояжера — одна из самых известных задач комбинаторной оптимизации, заключающаяся в поиске самого выгодного маршрута, проходящего через указанные города хотя бы по одному разу без или с последующим возвратом в исходный город. В условиях задачи также указывается критерий выгодности искомого маршрута, соответственно, возможно найти лучший маршрут.

Целью данной лабораторной работы является анализ и реализация двух алгоритмов решения задачи коммивояжера — алгоритма полного перебора и муравьиного алгоритма. Для достижения поставленной цели требуется решить следующие задачи:

— проанализировать указанные алгоритмы;

— реализовать указанные алгоритмы;

— провести исследование точности работы муравьиного алгоритма в зависимости от его входных параметров;

— описать и обосновать полученные результаты в отчете о выполненной лабораторной работе.

1 Аналитическая часть

В данном разделе будут рассмотрены алгоритмы решения задачи коммивояжера — алгоритм полного перебора и муравьиный алгоритм.

1.1 Алгоритм полного перебора

Алгоритм полного перебора заключается в последовательном переборе всех вариантов пути, проходящего по всем вершинам по одному разу, и выборе лучшего из этих путей. Для этого строятся все возможные последовательности обхода исходного графа. Из курса комбинаторики известно, что число перестановок n элементов, в данном случае — номеров вершин, равняется $n!$, что равняется числу проверяемых путей, поэтому алгоритмическую сложность алгоритма полного перебора можно описать как $O(n!)$.

1.2 Муравьиный алгоритм

Муравьиный алгоритм — это мета-эвристический роевой метод, предназначенный для решения полиномиальных задач комбинаторной оптимизации на графовой модели. Основываясь на наблюдениях реальной колонии муравьев, было выявлено, что муравьи способны находить кратчайший путь между гнездом и источником пищи. Кроме того, в то время, когда муравьи передвигаются от гнезда к пище и наоборот, они оставляют на своем пути уникальное вещество — феромон. Чем выше концентрация феромона у пути, тем выше вероятность того, что по нему пойдут последующие муравьи, соответственно, путь, с максимальной концентрацией феромона, и будет являться кратчайшим путем к пище [1].

Муравьиный алгоритм решает задачу по принципу поведения реальной муравьиной колонии. Первоначально, агенты (искусственные муравьи) помещаются в определенные вершины графа. Затем, каждый агент выполняет последовательность случайных перемещений от одной вершины к другой, в соответствие с задаваемой вероятностью или стратегией выбора пути. В общем случае, вероятность перехода по конкретному ребру зависит от его длины и концентрации феромона. После того, как агент посетит все вершины, происходит качественный анализ маршрута, а концентрация феромона, оставленного во время прохождения тура, обновляется согласно заданным

правилам, основанным на качестве пройденного пути [1].

При решении задачи коммивояжера оптимизируемая функция — длина всего маршрута. При завершении муравьями маршрутов производится выбор решения с наилучшим качеством, далее муравьи начинают выполнять описанную ранее процедуру заново, так продолжается несколько «дней» — итераций основного цикла [1]. Таким образом, алгоритмическую сложность муравьиного алгоритма можно описать как $O(t*m*n^2)$, где t – число дней, m – количество муравьев, n – количество вершин. В данном случае количество муравьев равно количеству вершин.

Муравьи имеют несколько характеристик:

— жадность – отвечает за желание ходить по более коротким дугам;

— память – запоминает пройденный маршрут;

— стадность – отвечает за желание идти по дуге с самым высоким уровнем феромона.

Вводится «привлекательность» дуги, обратно пропорциональная ее длине (1.1).

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

где D_{ij} — расстояние из текущего пункта i до заданного пункта j.

Формула определения вероятности перехода из i-й вершины в j-ю (1.2).

$$P_{kij} = \begin{cases} \frac{\tau_{ij}^a \eta_{ij}^b}{\sum_{q=1}^{m} \tau_{iq}^a \eta_{iq}^b}, \text{вершина не была посещена ранее муравьем k,} \\ 0, \text{иначе} \end{cases} \tag{1.2}$$

где a – коэффициент «жадности»(влияния длины пути), b – коэффициент «стадности»(влияние феромона), τ_{ij} – расстояния от города i до j, η_{ij} – количество феромонов на ребре ij.

После завершения движения всех муравьев, феромон обновляется(происходит испарение) по формуле (1.3):

$$\tau_{ij}(t+1) = (1-p)\tau_{ij}(t) + \Delta\tau_{ij}. \tag{1.3}$$

При этом

$$\Delta\tau_{ij} = \sum_{k=1}^{N} \tau_{ij}^{k}, \tag{1.4}$$

где

$$\Delta\tau_{ij}^{k} = \begin{cases} Q/L_k, \text{ребро посещено k-ым муравьем,} \\ 0, \text{иначе} \end{cases} \tag{1.5}$$

При этом вводится минимальный уровень феромона на дуге и при испарении выбирается максимум из этого минимального уровня и уровня, посчитанного по формуле. Таким образом сохраняется вероятность прохождения по всем дугам.

Путь выбирается по следующей схеме:

— каждый муравей имеет список запретов – список уже посещенных городов (вершин графа).

— муравьиная жадность – отвечает за желание посетить вершину.

— муравьиная стадность – отвечает за ощущение феромона на определенном ребре. При этом количество феромона на ребре в момент времени t обозначается как $\tau_{i,j}(t)$.

— по прохождении определенного пути муравей оставляет на нем некотрое количество феромона, которое показывает оптимальность сделанного выбора, это кол-во вычисляется по формуле (1.5)

Вывод

В данном разделе были рассмотрены решения задачи коммивояжера, а также их алгоритмические сложности.

2 Конструкторская часть

В данном разделе будут разработаны алгоритм полного перебора и муравьиный алгоритм.

2.1 Разработка алгоритмов

На вход каждому алгоритму подается матрица смежности исходного графа. По варианту граф полный, ориентированный (разная цена прохода по дуге в зависимости от направления).

На рисунке (2.1) приведен алгоритм решения задачи коммивояжера полным перебором, а на рисунок (2.2 — 2.6) приведена схема муравьиного алгоритма.

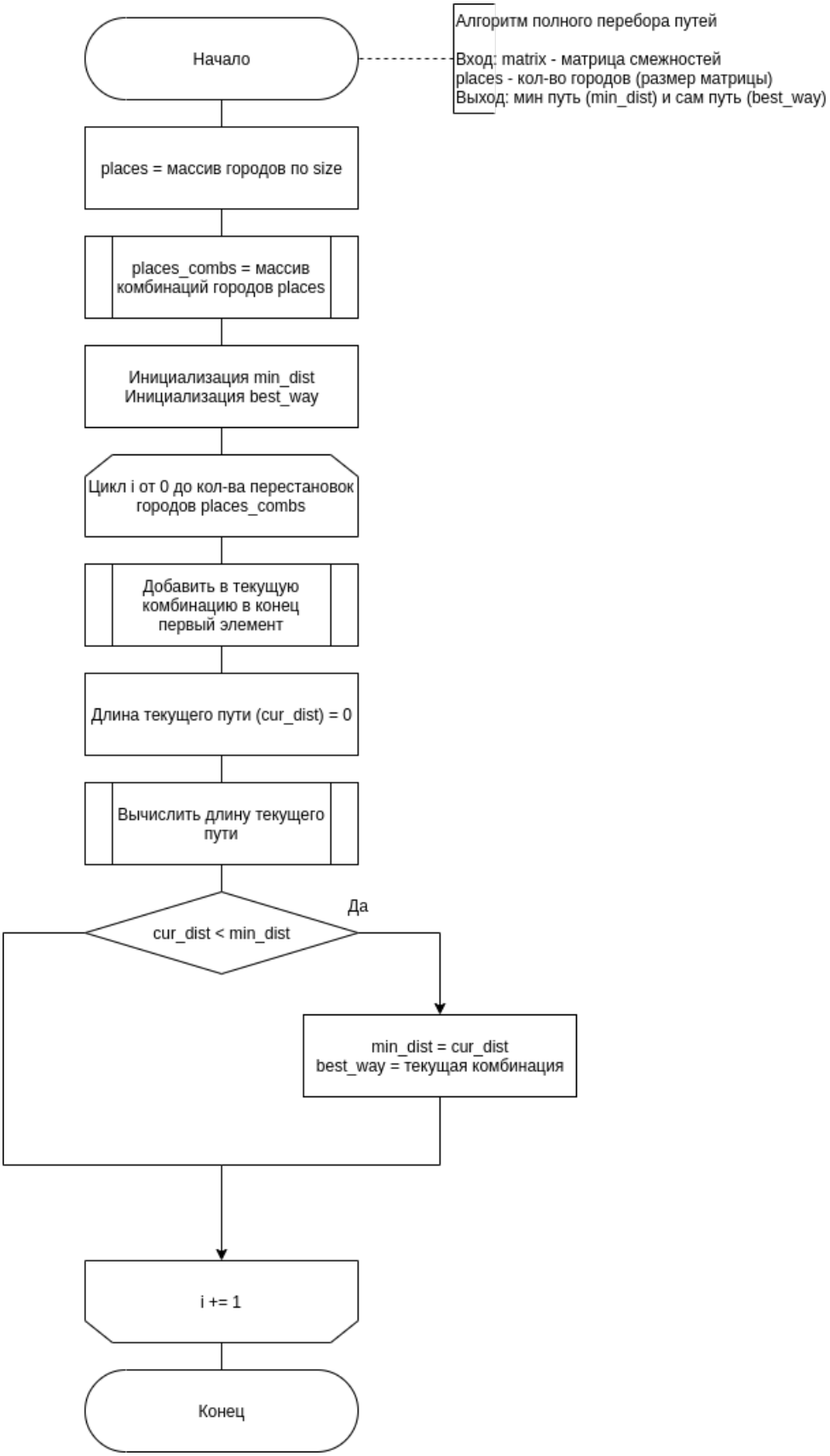

Рисунок 2.1 – Алгоритм полного перебора

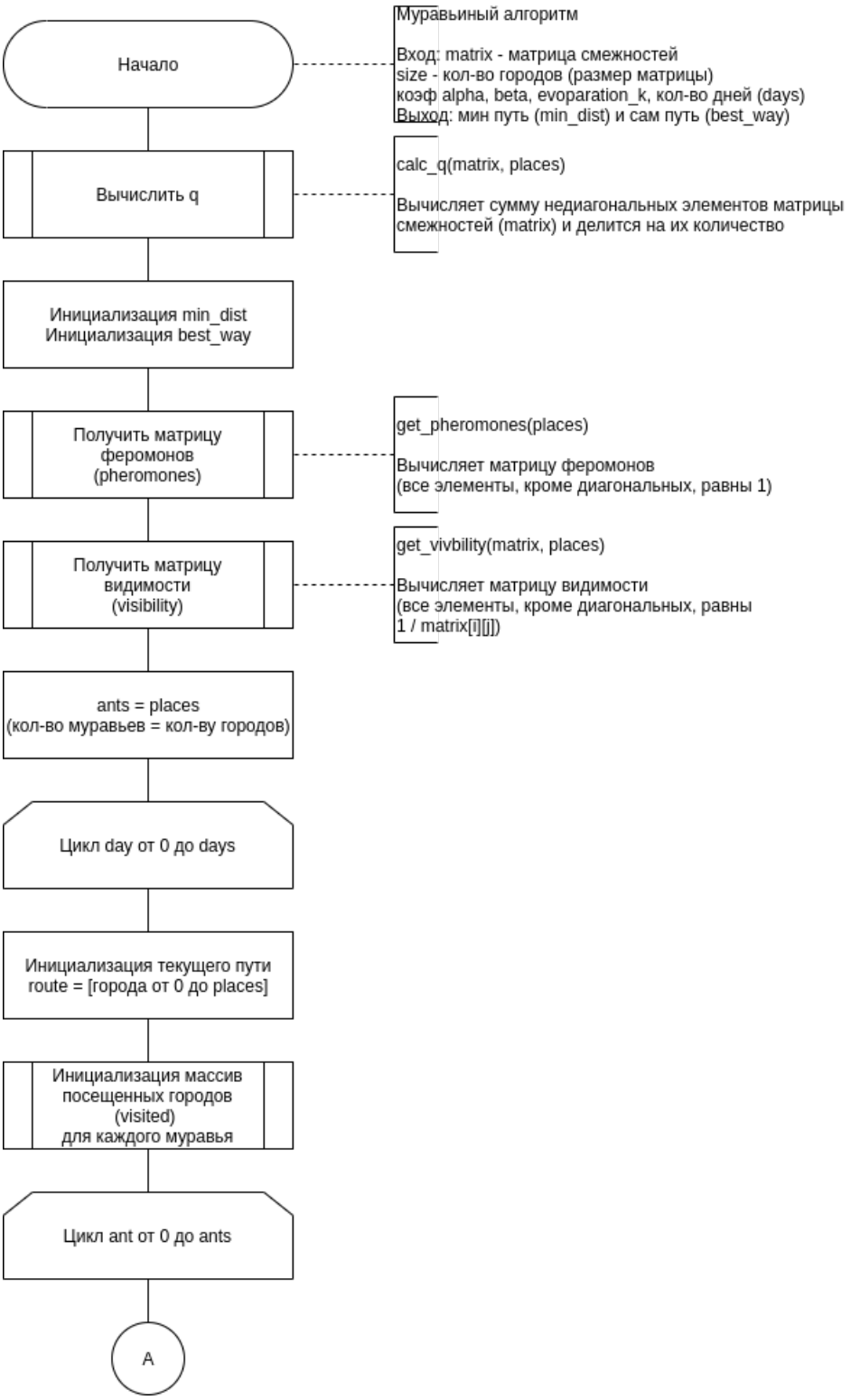

Рисунок 2.2 – Муравьиный алгоритм часть 1

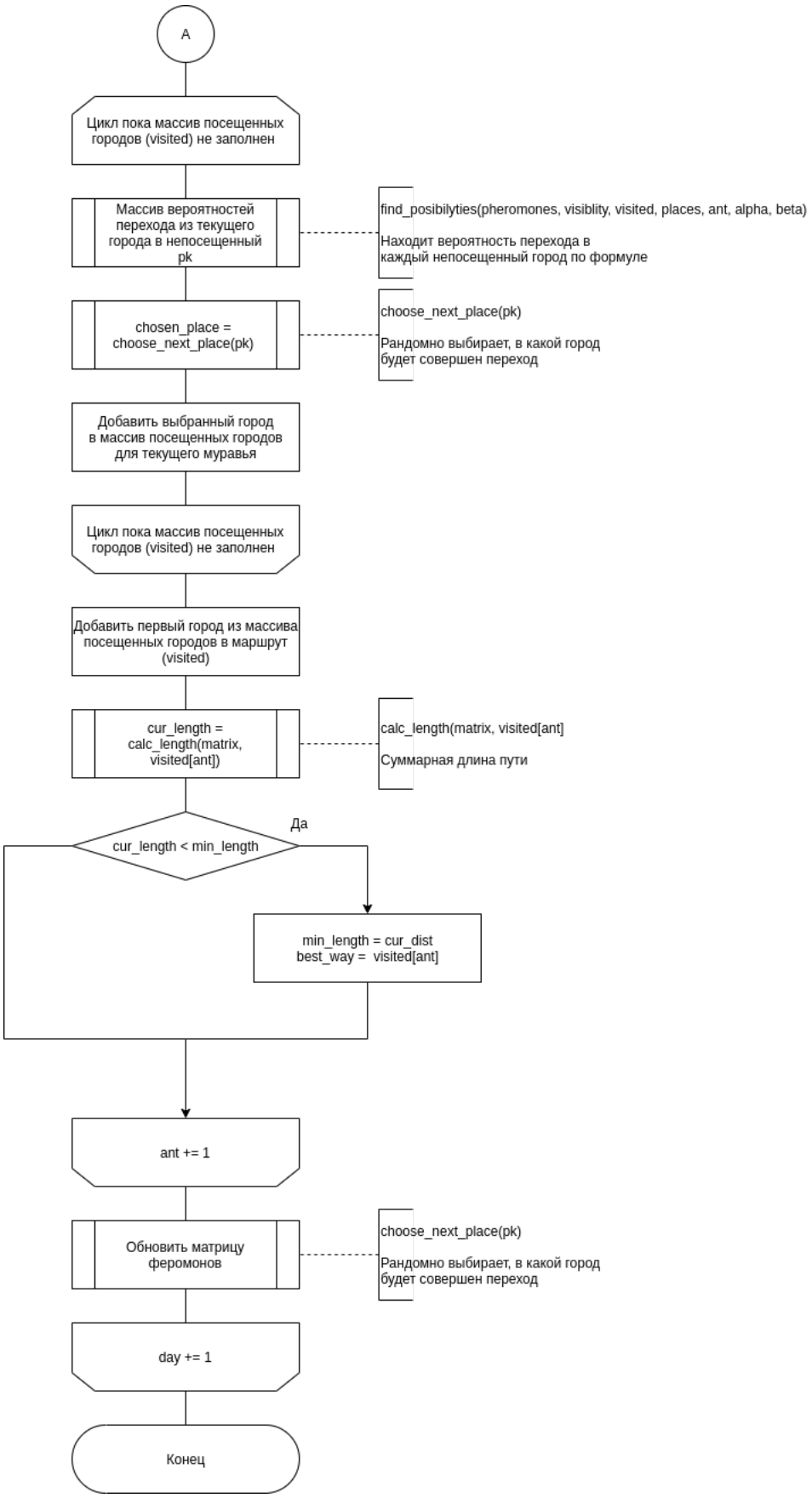

Рисунок 2.3 – Муравьиный алгоритм часть 2

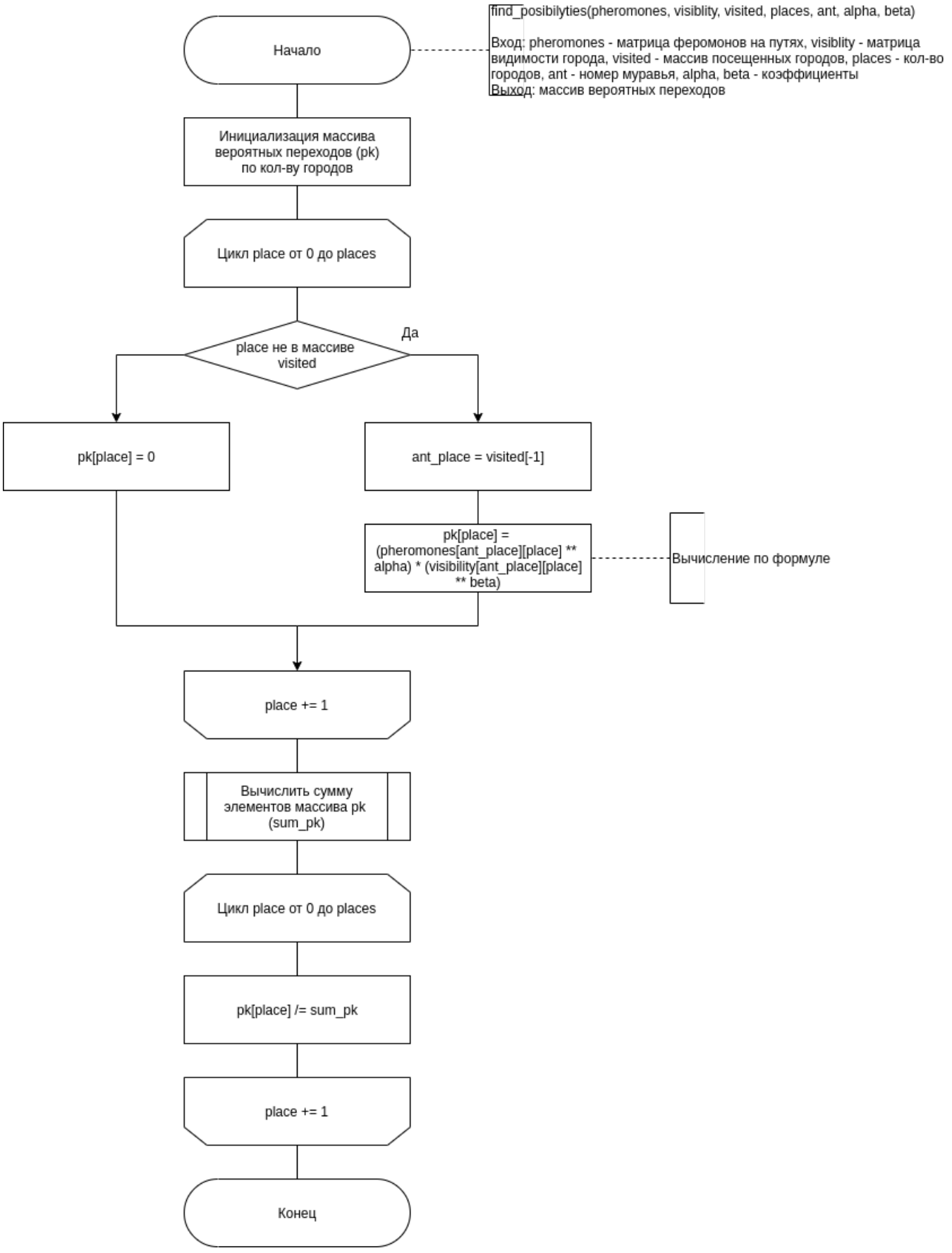

Рисунок 2.4 – Поиск вероятностей перехода в каждую из вершин

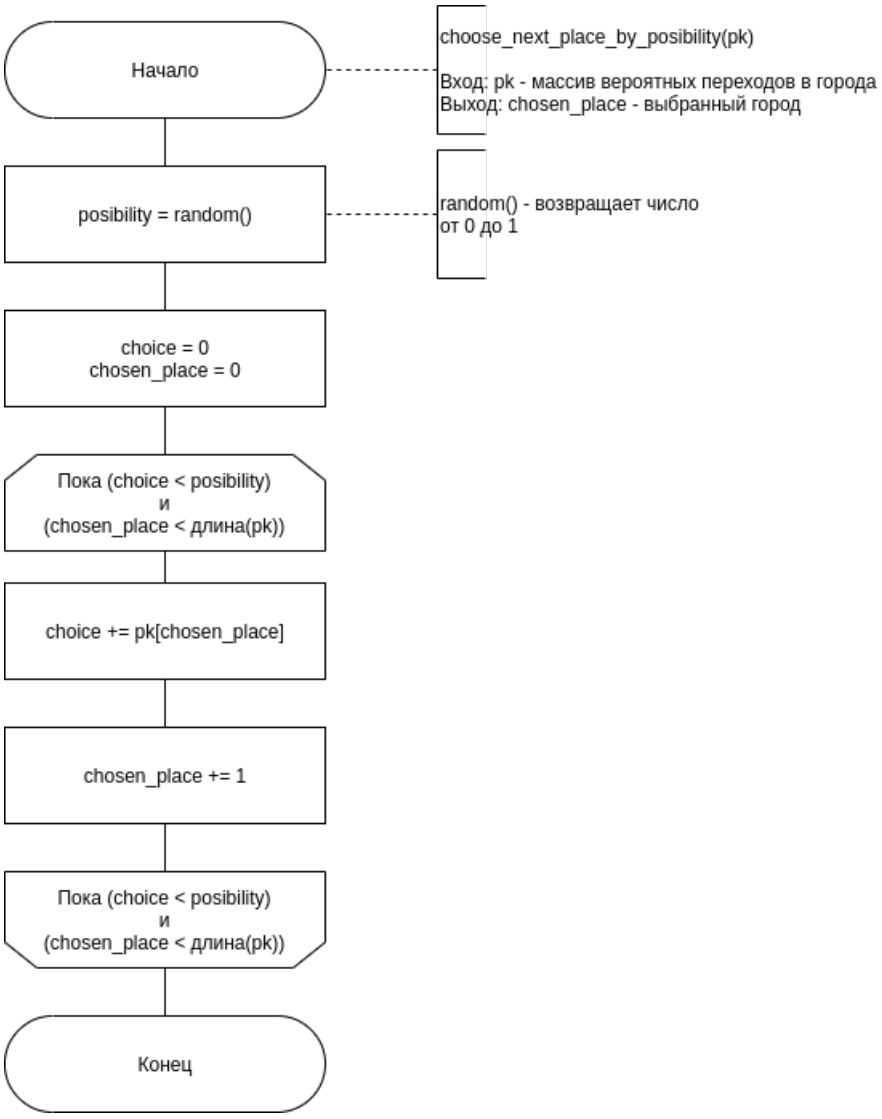

Рисунок 2.5 – Случайный выбор очередной вершины

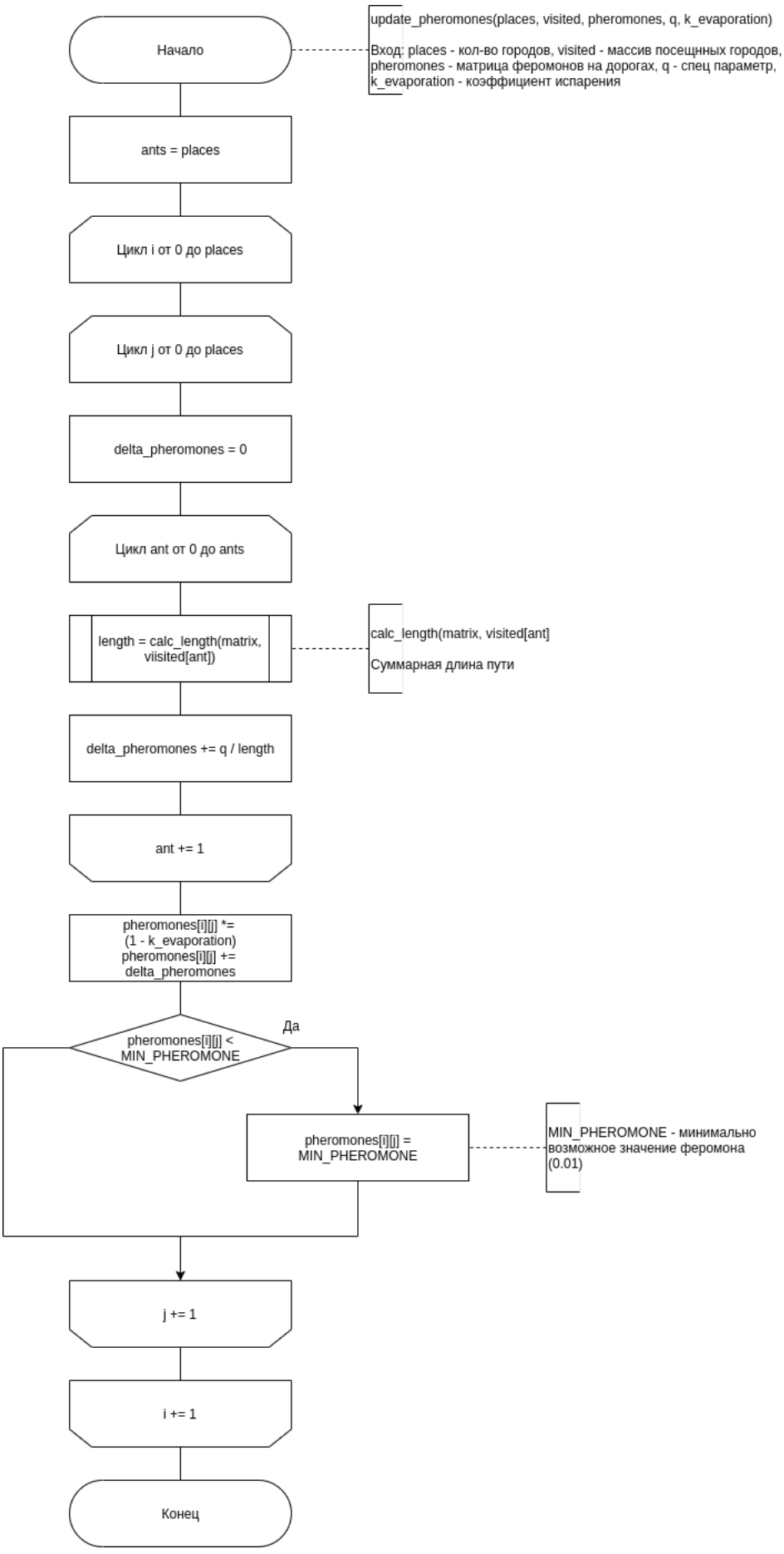

Рисунок 2.6 – Обновление феромона

Вывод

В данном разделе были разработаны алгоритмы решения задачи коммивояжера полным перебором и муравьиным алгоритмом.

3 Технологическая часть

В данном разделе будут приведены данные о выбранном языке программирования и коды алгоритмов.

3.1 Требования к ПО

Реализуемое ПО будет давать возможность ввести матрицу смежности исследуемого графа, а также все параметры муравьиного алгоритма.

3.2 Язык программирования

В данной работе для реализации алгоритмов был выбран язык программирования *Python* [2]. Язык предоставляет возможность работы с массивами и матрицами различной размерности, а также возможность создания пользовательского интерфейса.

3.3 Реализация алгоритмов

В листингах (1 — 3) представлены реализации алгоритмов полного перебора и муравьиного алгоритма.

Листинг 1 – Реализация алгоритма решения задачи коммивояжера полным перебором

```
def naive_algorithm(graph: list[list[int]]) -> (int, list[int]):
    if len(graph) == 0:
        return -1, None
    if len(graph[0]) != len(graph):
        return -1, None
    n = len(graph)
    options = list(permutations(range(n)))
    best_route = []
    best_summ = -1
    for option in options:
        summ = 0
        for i in range(len(option) - 1):
            if graph[option[i]][option[i + 1]] <= 0:
                summ = -1
                break
            summ += graph[option[i]][option[i + 1]]
        if summ == -1:
            continue
        if best_summ == -1:
            best_summ = summ
            best_route = option
        elif summ < best_summ:
            best_summ = summ
            best_route = option
    return best_summ, list(best_route)
```

Листинг 2 – Функция обхода графа одним муравьем

```
def ant_search(graph: list[list[int]], pheromone_matrix:
    list[list[float]],
                greed: float, herding: float, start_point: int) ->
                    (int, list[int]):
    n = len(graph)
    visited = {start_point}
    cur_point = start_point
    sum_route = 0
    route = [start_point]
    while len(visited) < n:
        probabilities = []
        sum_not_visited = 0.0
        for i in range(n):
            if i not in visited:
                sum_not_visited += (((1.0 / graph[cur_point][i]) **
                    greed) * (pheromone_matrix[cur_point][i] **
                    herding))
        for i in range(n):
            if i in visited:
                probabilities.append(0.0)
            else:
                upper = (((1.0 / graph[cur_point][i]) ** greed) *
                    (pheromone_matrix[cur_point][i] ** herding))
                probabilities.append(upper / sum_not_visited)
        epsilon = random()
        sum_prob = 0.0
        ind = 0
        for i in range(n):
            sum_prob += probabilities[i]
            if sum_prob >= epsilon and i not in visited:
                ind = i
                break
        sum_route += graph[cur_point][ind]
        cur_point = ind
        visited.add(cur_point)
        route.append(cur_point)
    return sum_route, list(route)
```

Листинг 3 – Реализация решения задачи коммивояжера муравьиным алгоритмом

```
def ant_search(graph: list[list[int]], pheromone_matrix:
    list[list[float]],
                greed: float, herding: float, start_point: int) ->
                    (int, list[int]):
    n = len(graph)
    visited = {start_point}
    cur_point = start_point
    sum_route = 0
    route = [start_point]
    while len(visited) < n:
        probabilities = []
        sum_not_visited = 0.0
        for i in range(n):
            if i not in visited:
                sum_not_visited += (((1.0 / graph[cur_point][i]) **
                    greed) * (pheromone_matrix[cur_point][i] **
                    herding))
        for i in range(n):
            if i in visited:
                probabilities.append(0.0)
            else:
                upper = (((1.0 / graph[cur_point][i]) ** greed) *
                    (pheromone_matrix[cur_point][i] ** herding))
                probabilities.append(upper / sum_not_visited)
        epsilon = random()
        sum_prob = 0.0
        ind = 0
        for i in range(n):
            sum_prob += probabilities[i]
            if sum_prob >= epsilon and i not in visited:
                ind = i
                break
        sum_route += graph[cur_point][ind]
        cur_point = ind
        visited.add(cur_point)
        route.append(cur_point)
    return sum_route, list(route)
```

Вывод

В данном разделе были приведены коды реализаций алгоритмов решения задачи коммивояжера.

4 Исследовательская часть

В данном разделе будет приведен сравнительный анализ алгоритмов при различных параметрах муравьиного алгоритма, а также замеры времени работы алгоритмов для матриц различного размера.

4.1 Технические характеристики

В данном разделе представлены технические характеристики устройства, на котором проводилось исследование.

— операционная система: Windows11 Домашняя, версия 23H2, сборка ОС 22631.4037;

— оперативная память: 16 ГБ;

— процессор: 13th Gen Intel(R) Core(TM) i5-13500H 2.60 ГГц.

При тестировании ноутбук был включен в сеть электропитания и нагружен только встроенными приложениями окружения, а также системой тестирования.

4.2 Замеры времени

Для замеров времени использовалась функция *process_time*() из библиотеки *time* языка *Python*. Функция возвращает процессорное время в секундах.

На рисунке (4.1) представлен график замеров времени алгоритмом полного перебора и муравьиным алгоритмом для размерностей матрицы от 1x1 до 10x10:

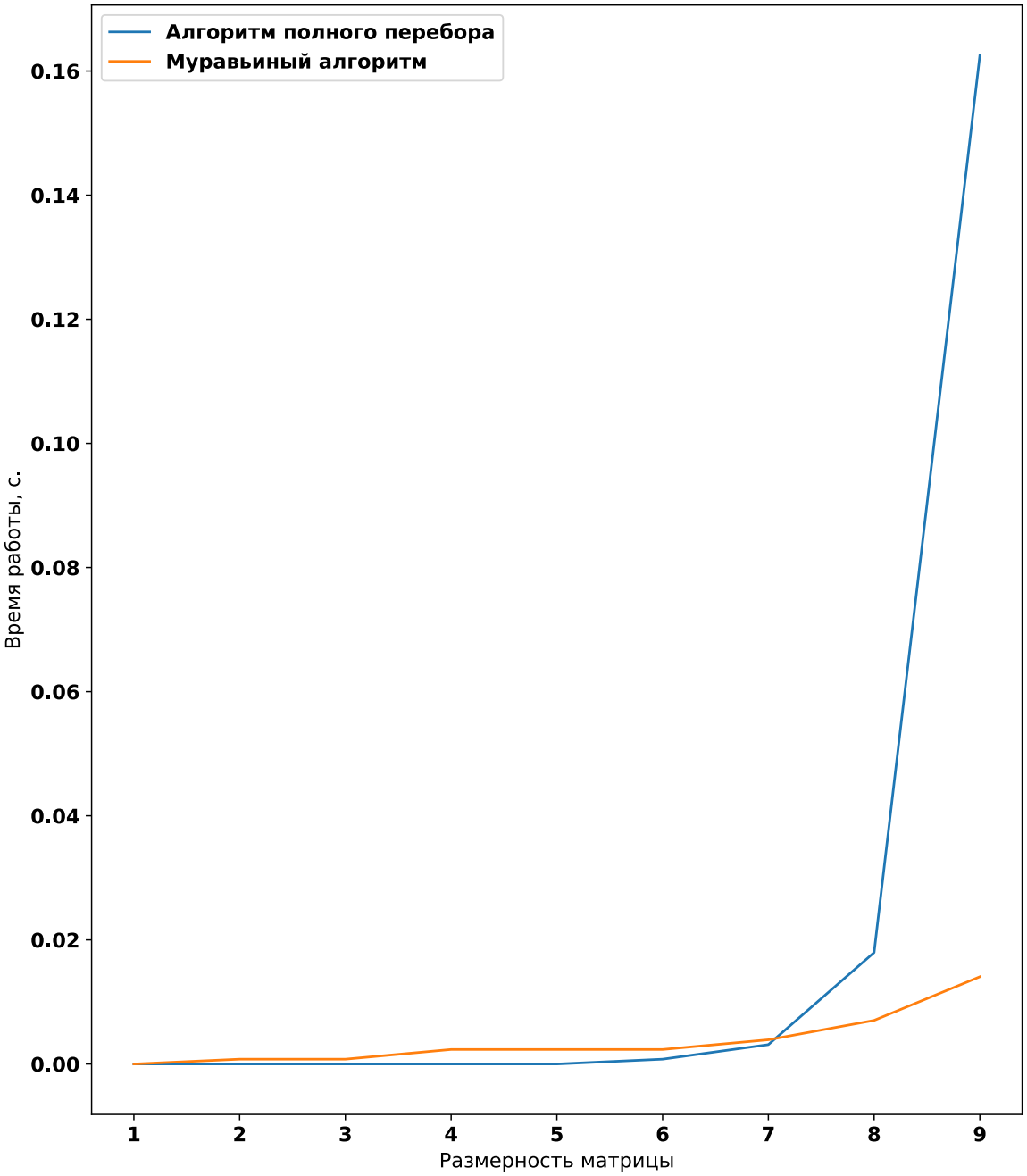

Рисунок 4.1 – График измерений

4.3 Постановка исследования

Исследование проводилось на двух различных матрицах размерами 4x4 и 8x8. Итоговая таблица будет содержать значения параметров муравьиного алгоритма, результат определения длины кратчайшего пути муравьиным алгоритмом, а также ошибку — разность длин путей (за эталон принимается длина пути, подсчитанная наивным алгоритмом, так как он является точным).

Обозначения столбцов:

— *Greed* — коэффициент жадности;

— *Evaporation* — коэффициент испарения;

— *Days* — количество дней жизни колонии муравьев;

— *Result* — эталонный результат, полученный методом полного перебора (наивным алгоритмом);

— *Mistake* — значение ошибки (отклонения длины пути, подсчитанной муравьиным алгоритмом, от длины пути, подсчитанной полным перебором).

Параметры муравьиного алгоритма варьировались от 0.1 до 0.9 с шагом 0.1.

4.3.1 Исследование для первой матрицы

Первая матрица:

$$K_1 = \begin{pmatrix} 0 & 2 & 7 & 10 \\ 16 & 0 & 9 & 12 \\ 1 & 11 & 0 & 16 \\ 7 & 12 & 12 & 0 \end{pmatrix} \tag{4.1}$$

Результаты для количества дней, равного 10:

Таблица 4.1 – Выжимка из результатов

Days	*Greed*	*Evaporation*	*Result*	*Mistake*
10	0.1	0.1	15	0
10	0.1	0.25	15	0
10	0.1	0.5	15	0
10	0.1	0.75	15	0
10	0.1	0.9	15	0
10	0.25	0.1	15	0
10	0.25	0.25	15	0
10	0.25	0.5	15	0
10	0.25	0.75	15	0
10	0.25	0.9	15	0
10	0.5	0.1	15	0
10	0.5	0.25	15	0
10	0.5	0.5	15	0
10	0.5	0.75	15	0
10	0.5	0.9	15	0
10	0.75	0.1	15	0
10	0.75	0.25	15	0
10	0.75	0.5	15	0
10	0.75	0.75	15	0
10	0.75	0.9	15	0
10	0.9	0.1	15	0
10	0.9	0.25	15	0
10	0.9	0.5	15	0
10	0.9	0.75	15	0
10	0.9	0.9	15	0

Остальные результаты представлены в Приложении А.

4.4 Исследование для второй матрицы

Вторая матрица содержит в себе на порядок большие числа, чем первая:

$$K_1 = \begin{pmatrix} 0 & 3 & 15 & 18 & 6 & 15 & 18 & 11 \\ 19 & 0 & 9 & 15 & 1 & 12 & 18 & 5 \\ 15 & 4 & 0 & 16 & 2 & 1 & 18 & 15 \\ 16 & 12 & 1 & 0 & 11 & 2 & 17 & 8 \\ 16 & 7 & 17 & 20 & 0 & 3 & 19 & 2 \\ 16 & 15 & 2 & 17 & 17 & 0 & 20 & 3 \\ 10 & 15 & 1 & 17 & 9 & 17 & 0 & 16 \\ 12 & 1 & 11 & 2 & 13 & 9 & 20 & 0 \end{pmatrix} \tag{4.2}$$

Результаты для количества дней, равного 10:

Таблица 4.2 – Выжимка из результатов

Days	*Greed*	*Evaporation*	*Result*	*Mistake*
10	0.1	0.1	27	7
10	0.1	0.25	28	8
10	0.1	0.5	30	10
10	0.1	0.75	22	2
10	0.1	0.9	43	23
10	0.25	0.1	28	8
10	0.25	0.25	20	0
10	0.25	0.5	27	7
10	0.25	0.75	36	16
10	0.25	0.9	26	6
10	0.5	0.1	20	0
10	0.5	0.25	20	0
10	0.5	0.5	28	8
10	0.5	0.75	20	0
10	0.5	0.9	20	0
10	0.75	0.1	27	7
10	0.75	0.25	30	10
10	0.75	0.5	26	6
10	0.75	0.75	27	7
10	0.75	0.9	20	0
10	0.9	0.1	20	0
10	0.9	0.25	28	8
10	0.9	0.5	27	7
10	0.9	0.75	26	6
10	0.9	0.9	30	10

Остальные результаты представлены в Приложении Б.

Вывод

В результате исследования получено, что использование муравьиного алгоритма наиболее эффективно на больших размерах матриц. По результатам исследования был сделан вывод, что уже при размерах матрицы от 8x8 муравьиный алгоритм работает существенно быстрее алгоритма полного перебора, однако нужно учитывать погрешность муравьиного алгоритма, так как он далеко не всегда находит путь наименьшей длины, который позволяет найти алгоритм полного перебора (таблица (4.2)).

При проведении исследования точности работы муравьиного алгоритма было определено, что на первой матрице алгоритм практически всегда давал точный результат из-за малого размера матрицы. При этом для матрицы большей размерности были определены параметры, на которых алгоритм показывает лучшие результаты:

— $greed = 0.5$, $evaporation \in [0.1, 0.25, 0.75, 0.9]$;

— $greed = 0.25$, $evaporation \in [0.1, 0.25, 0.5]$;

при этом первый набор параметров позволяет точно определить длину кратчайшего пути всего за 10 дней, в то время как второй набор параметров — за 80.

ЗАКЛЮЧЕНИЕ

В результате выполнения исследования было определено, что муравьиный алгоритм стоит использовать при больших размерах входных матриц, уже при размерах от 8x8 он дает существенно более высокую скорость решения задачи, чем алгоритм полного перебора. При этом стоит учитывать, что муравьиный алгоритм может давать погрешности, однако при увеличении количества «дней жизни» муравьиной колонии эти погрешности уменьшаются.

В ходе выполнения работы решены следующие задачи:

— проанализированы алгоритмы решения задачи коммивояжера;

— реализованы указанные алгоритмы;

— проведено исследование времени работы алгоритмов и точности работы муравьиного алгоритма в зависимости от его входных параметров;

— описаны и обоснованы полученные результаты в отчете о выполненной лабораторной работе.

Поставленная цель достигнута, все задачи решены.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] В.М. Курейчик, А.В. Мартынов ОБ АЛГОРИТМАХ РЕШЕНИЯ ЗАДАЧИ КОММИВОЯЖЕРА С ВРЕМЕННЫМИ ОГРАНИЧЕНИЯМИ [Электронный ресурс]. Режим доступа: https://ivtio.sfedu.ru/lib/15/1-1(15)2014.pdf

[2] Документация языка Welcome to Python [Электронный ресурс]. Режим доступа: https://www.python.org (дата обращения: 30.09.2024).

[3] Документация функции *proccess_time* [Электронный ресурс]. Режим доступа: https://docs.python.org/3/library/time.html#time.process_time

5 Приложение А

Таблица 5.1 – Результаты для первой матрицы

Days	*Greed*	*Evaporation*	*Result*	*Mistake*
10	0.1	0.1	15	0
10	0.1	0.25	15	0
10	0.1	0.5	15	0
10	0.1	0.75	15	0
10	0.1	0.9	15	0
10	0.25	0.1	15	0
10	0.25	0.25	15	0
10	0.25	0.5	15	0
10	0.25	0.75	15	0
10	0.25	0.9	15	0
10	0.5	0.1	15	0
10	0.5	0.25	15	0
10	0.5	0.5	15	0
10	0.5	0.75	15	0
10	0.5	0.9	15	0
10	0.75	0.1	15	0
10	0.75	0.25	15	0
10	0.75	0.5	15	0
10	0.75	0.75	15	0
10	0.75	0.9	15	0
10	0.9	0.1	15	0
10	0.9	0.25	15	0
10	0.9	0.5	15	0
10	0.9	0.75	15	0
10	0.9	0.9	15	0
20	0.1	0.1	15	0
20	0.1	0.25	15	0
20	0.1	0.5	15	0
20	0.1	0.75	15	0

20	0.1	0.9	15	0
20	0.25	0.1	15	0
20	0.25	0.25	15	0
20	0.25	0.5	15	0
20	0.25	0.75	15	0
20	0.25	0.9	15	0
20	0.5	0.1	15	0
20	0.5	0.25	15	0
20	0.5	0.5	15	0
20	0.5	0.75	15	0
20	0.5	0.9	15	0
20	0.75	0.1	15	0
20	0.75	0.25	15	0
20	0.75	0.5	15	0
20	0.75	0.75	15	0
20	0.75	0.9	15	0
20	0.9	0.1	15	0
20	0.9	0.25	15	0
20	0.9	0.5	15	0
20	0.9	0.75	15	0
20	0.9	0.9	15	0
40	0.1	0.1	15	0
40	0.1	0.25	15	0
40	0.1	0.5	15	0
40	0.1	0.75	15	0
40	0.1	0.9	15	0
40	0.25	0.1	15	0
40	0.25	0.25	15	0
40	0.25	0.5	15	0
40	0.25	0.75	15	0
40	0.25	0.9	15	0
40	0.5	0.1	15	0
40	0.5	0.25	15	0

40	0.5	0.5	15	0
40	0.5	0.75	15	0
40	0.5	0.9	15	0
40	0.75	0.1	15	0
40	0.75	0.25	15	0
40	0.75	0.5	15	0
40	0.75	0.75	15	0
40	0.75	0.9	15	0
40	0.9	0.1	15	0
40	0.9	0.25	15	0
40	0.9	0.5	15	0
40	0.9	0.75	15	0
40	0.9	0.9	15	0
80	0.1	0.1	15	0
80	0.1	0.25	15	0
80	0.1	0.5	15	0
80	0.1	0.75	15	0
80	0.1	0.9	15	0
80	0.25	0.1	15	0
80	0.25	0.25	15	0
80	0.25	0.5	15	0
80	0.25	0.75	15	0
80	0.25	0.9	15	0
80	0.5	0.1	15	0
80	0.5	0.25	15	0
80	0.5	0.5	15	0
80	0.5	0.75	15	0
80	0.5	0.9	15	0
80	0.75	0.1	15	0
80	0.75	0.25	15	0
80	0.75	0.5	15	0
80	0.75	0.75	15	0
80	0.75	0.9	15	0

80	0.9	0.1	15	0
80	0.9	0.25	15	0
80	0.9	0.5	15	0
80	0.9	0.75	15	0
80	0.9	0.9	15	0
160	0.1	0.1	15	0
160	0.1	0.25	15	0
160	0.1	0.5	15	0
160	0.1	0.75	15	0
160	0.1	0.9	15	0
160	0.25	0.1	15	0
160	0.25	0.25	15	0
160	0.25	0.5	15	0
160	0.25	0.75	15	0
160	0.25	0.9	15	0
160	0.5	0.1	15	0
160	0.5	0.25	15	0
160	0.5	0.5	15	0
160	0.5	0.75	15	0
160	0.5	0.9	15	0
160	0.75	0.1	15	0
160	0.75	0.25	15	0
160	0.75	0.5	15	0
160	0.75	0.75	15	0
160	0.75	0.9	15	0
160	0.9	0.1	15	0
160	0.9	0.25	15	0
160	0.9	0.5	15	0
160	0.9	0.75	15	0
160	0.9	0.9	15	0
320	0.1	0.1	15	0
320	0.1	0.25	15	0
320	0.1	0.5	15	0

320	0.1	0.75	15	0
320	0.1	0.9	15	0
320	0.25	0.1	15	0
320	0.25	0.25	15	0
320	0.25	0.5	15	0
320	0.25	0.75	15	0
320	0.25	0.9	15	0
320	0.5	0.1	15	0
320	0.5	0.25	15	0
320	0.5	0.5	15	0
320	0.5	0.75	15	0
320	0.5	0.9	15	0
320	0.75	0.1	15	0
320	0.75	0.25	15	0
320	0.75	0.5	15	0
320	0.75	0.75	15	0
320	0.75	0.9	15	0
320	0.9	0.1	15	0
320	0.9	0.25	15	0
320	0.9	0.5	15	0
320	0.9	0.75	15	0
320	0.9	0.9	15	0

6 Приложение Б

Таблица 6.1 – Результаты для второй матрицы

Days	*Greed*	*Evaporation*	*Result*	*Mistake*
10	0.1	0.1	27	7
10	0.1	0.25	28	8
10	0.1	0.5	30	10
10	0.1	0.75	22	2
10	0.1	0.9	43	23
10	0.25	0.1	28	8
10	0.25	0.25	20	0
10	0.25	0.5	27	7
10	0.25	0.75	36	16
10	0.25	0.9	26	6
10	0.5	0.1	20	0
10	0.5	0.25	20	0
10	0.5	0.5	28	8
10	0.5	0.75	20	0
10	0.5	0.9	20	0
10	0.75	0.1	27	7
10	0.75	0.25	30	10
10	0.75	0.5	26	6
10	0.75	0.75	27	7
10	0.75	0.9	20	0
10	0.9	0.1	20	0
10	0.9	0.25	28	8
10	0.9	0.5	27	7
10	0.9	0.75	26	6
10	0.9	0.9	30	10
20	0.1	0.1	20	0
20	0.1	0.25	22	2
20	0.1	0.5	28	8
20	0.1	0.75	31	11

20	0.1	0.9	23	3
20	0.25	0.1	30	10
20	0.25	0.25	22	2
20	0.25	0.5	37	17
20	0.25	0.75	27	7
20	0.25	0.9	20	0
20	0.5	0.1	27	7
20	0.5	0.25	22	2
20	0.5	0.5	26	6
20	0.5	0.75	30	10
20	0.5	0.9	35	15
20	0.75	0.1	23	3
20	0.75	0.25	30	10
20	0.75	0.5	22	2
20	0.75	0.75	28	8
20	0.75	0.9	26	6
20	0.9	0.1	26	6
20	0.9	0.25	28	8
20	0.9	0.5	30	10
20	0.9	0.75	28	8
20	0.9	0.9	28	8
40	0.1	0.1	22	2
40	0.1	0.25	22	2
40	0.1	0.5	26	6
40	0.1	0.75	26	6
40	0.1	0.9	28	8
40	0.25	0.1	20	0
40	0.25	0.25	26	6
40	0.25	0.5	27	7
40	0.25	0.75	22	2
40	0.25	0.9	37	17
40	0.5	0.1	27	7
40	0.5	0.25	20	0

40	0.5	0.5	22	2
40	0.5	0.75	31	11
40	0.5	0.9	30	10
40	0.75	0.1	23	3
40	0.75	0.25	36	16
40	0.75	0.5	26	6
40	0.75	0.75	28	8
40	0.75	0.9	26	6
40	0.9	0.1	30	10
40	0.9	0.25	27	7
40	0.9	0.5	26	6
40	0.9	0.75	27	7
40	0.9	0.9	30	10
80	0.1	0.1	20	0
80	0.1	0.25	22	2
80	0.1	0.5	20	0
80	0.1	0.75	30	10
80	0.1	0.9	33	13
80	0.25	0.1	20	0
80	0.25	0.25	20	0
80	0.25	0.5	20	0
80	0.25	0.75	28	8
80	0.25	0.9	36	16
80	0.5	0.1	22	2
80	0.5	0.25	20	0
80	0.5	0.5	26	6
80	0.5	0.75	26	6
80	0.5	0.9	22	2
80	0.75	0.1	22	2
80	0.75	0.25	28	8
80	0.75	0.5	22	2
80	0.75	0.75	20	0
80	0.75	0.9	28	8

80	0.9	0.1	27	7
80	0.9	0.25	27	7
80	0.9	0.5	30	10
80	0.9	0.75	27	7
80	0.9	0.9	26	6
160	0.1	0.1	20	0
160	0.1	0.25	20	0
160	0.1	0.5	27	7
160	0.1	0.75	20	0
160	0.1	0.9	26	6
160	0.25	0.1	22	2
160	0.25	0.25	20	0
160	0.25	0.5	23	3
160	0.25	0.75	29	9
160	0.25	0.9	26	6
160	0.5	0.1	20	0
160	0.5	0.25	20	0
160	0.5	0.5	22	2
160	0.5	0.75	30	10
160	0.5	0.9	30	10
160	0.75	0.1	26	6
160	0.75	0.25	20	0
160	0.75	0.5	27	7
160	0.75	0.75	30	10
160	0.75	0.9	28	8
160	0.9	0.1	30	10
160	0.9	0.25	22	2
160	0.9	0.5	30	10
160	0.9	0.75	30	10
160	0.9	0.9	30	10
320	0.1	0.1	20	0
320	0.1	0.25	20	0
320	0.1	0.5	20	0

320	0.1	0.75	30	10
320	0.1	0.9	36	16
320	0.25	0.1	20	0
320	0.25	0.25	26	6
320	0.25	0.5	20	0
320	0.25	0.75	34	14
320	0.25	0.9	20	0
320	0.5	0.1	26	6
320	0.5	0.25	27	7
320	0.5	0.5	26	6
320	0.5	0.75	30	10
320	0.5	0.9	28	8
320	0.75	0.1	26	6
320	0.75	0.25	28	8
320	0.75	0.5	27	7
320	0.75	0.75	30	10
320	0.75	0.9	27	7
320	0.9	0.1	27	7
320	0.9	0.25	30	10
320	0.9	0.5	28	8
320	0.9	0.75	30	10
320	0.9	0.9	28	8