Отчет по лабораторной работе №6
по курсу "Анализ алгоритмов"
по теме "Муравьиные алгоритмы"

Студент: Доктор А.А. ИУ7-53
Преподаватель: Волкова Л.Л., Строганов Ю.В.

2018 г.

Содержание

Введение

Муравьиный алгоритм (алгоритм оптимизации подражанием муравьиной колонии, англ. ant colony optimization, ACO) — один из эффективных полиномиальныхалгоритмов для нахождения приближённых решений задачи коммивояжёра, а также решения аналогичных задач поиска маршрутов на графах. Суть подхода заключается в анализе и использовании модели поведения муравьёв, ищущих пути от колонии к источнику питания и представляет собой метаэвристическую оптимизацию. Первая версия алгоритма, предложенная доктором наук Марко Дориго в 1992 году, была направлена на поиск оптимального пути в графе[1].

В реальном мире муравьи (первоначально) ходят в случайном порядке и по нахождении продовольствия возвращаются в свою колонию, прокладывая феромонами тропы. Если другие муравьи находят такие тропы, они, вероятнее всего, пойдут по ним. Вместо того, чтобы отслеживать цепочку, они укрепляют её при возвращении, если в конечном итоге находят источник питания. Со временем феромонная тропа начинает испаряться, тем самым уменьшая свою привлекательную силу. Чем больше времени требуется для прохождения пути до цели и обратно, тем сильнее испарится феромонная тропа. На коротком пути, для сравнения, прохождение будет более быстрым и как следствие, плотность феромонов остаётся высокой. Испарение феромонов также имеет свойство избежания стремления к локально-оптимальному решению. Если бы феромоны не испарялись, то путь, выбранный первым, был бы самым привлекательным. В этом случае, исследования пространственных решений были бы ограниченными. Таким образом, когда один муравей находит (например, короткий) путь от колонии до источника пищи, другие муравьи, скорее всего пойдут по этому пути, и положительные отзывы в конечном итоге приводят всех муравьёв к одному, кратчайшему, пути[2]. На рис. 1 продемонстрирован принцип устройства муравьиного подхода[3].

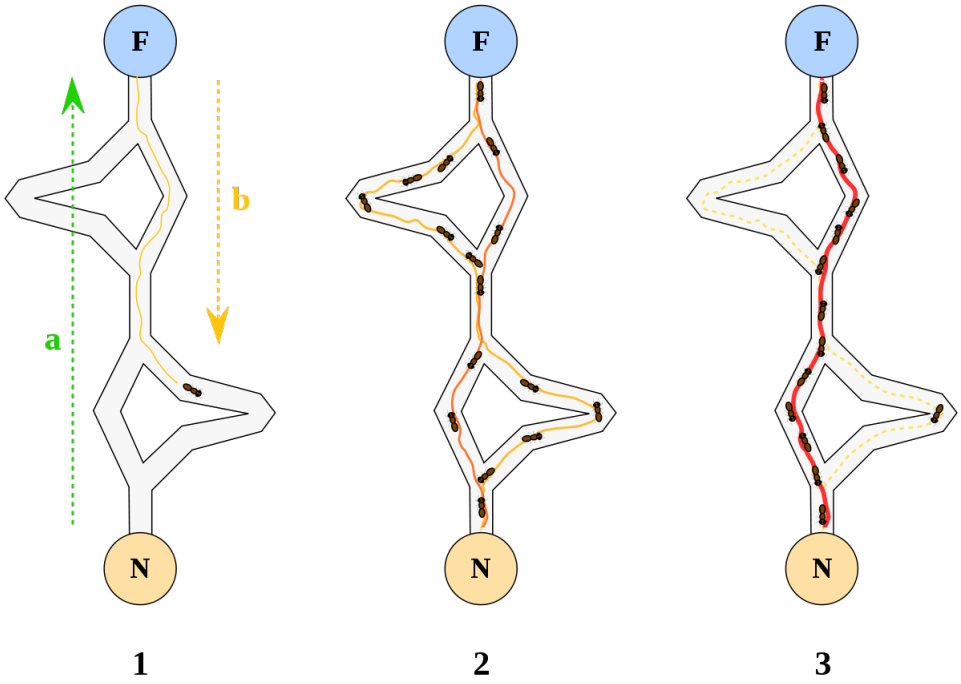

Рис. 1: Принцип работы муравьиного алгоритма

Задача коммивояжёра (англ. Travelling salesman problem, сокращённо TSP) — одна из самых известных задач комбинаторной оптимизации, заключающаяся в поиске самого выгодного маршрута, проходящего через указанные города хотя бы по одному разу с последующим возвратом в исходный город. В условиях задачи указываются критерий выгодности маршрута (кратчайший, самый дешёвый, совокупный критерий и тому подобное) и соответствующие матрицы расстояний, стоимости и тому подобного. Как правило, указывается, что маршрут должен проходить через каждый город только один раз — в таком случае выбор осуществляется среди гамильтоновых циклов. Существует несколько частных случаев общей постановки задачи, в частности, геометрическая задача коммивояжёра (также называемая планарной или евклидовой, когда матрица расстояний отражает расстояния между точками на плоскости), метрическая задача коммивояжёра (когда на матрице стоимостей выполняется неравенство треугольника), симметричная и асимметричная задачи коммивояжёра. Также существует обобщение задачи, так называемая обобщённая задача коммивояжёра.

Оптимизационная постановка задачи относится к классу NP-трудных задач, впрочем, как и большинство её частных случаев. Версия «decision problem» (то есть такая, в которой ставится вопрос, существует ли маршрут не длиннее, чем заданное значение k) относится к классу NP-полных задач. Задача коммивояжёра относится к числу трансвычислительных: уже при относительно небольшом числе городов (66 и более) она не может быть решена методом перебора вариантов никакими теоретически мыслимыми

компьютерами за время, меньшее нескольких миллиардов лет.

На рис. 2 представлен оптимальный маршрут коммивояжёра через 15 крупнейших городов Германии. Указанный маршрут является самым коротким из всех возможных 43 589 145 600 вариантов.

Рис. 2: оптимальный маршрут коммивояжёра через 15 крупнейших городов Германии

Задачи работы

В ходе выполнения данной лабораторной работы, мной были выполнены следующие задачи:

1) С помощью одного из муравьиных алгоритмов решить задачу комми-вояжёра;

2) провести замеры скорости работы работы алгоритма при разных коэффициентах(влияния расстоянийеромонов), количестве поколений и элитных муравьев.

1 Аналитическая часть

В данном разделе приведено описание оригинального муравьиного алгоритма и несколько способов его улучшения

1.1 Оригинальный муравьиеый алгоритм

Каждый муравей хранит в памяти список пройденных им узлов. Этот список называют списком запретов (tabu list) или просто памятью муравья. Выбирая узел для следующего шага, муравей «помнит» об уже пройденных узлах и не рассматривает их в качестве возможных для перехода. На каждом шаге список запретов пополняется новым узлом, а перед новой итерацией алгоритма – то есть перед тем, как муравей вновь проходит путь – он опустошается.

Кроме списка запретов, при выборе узла для перехода муравей руководствуется «привлекательностью» ребер, которые он может пройти. Она зависит, во-первых, от расстояния между узлами (то есть от веса ребра), а во-вторых, от следов феромонов, оставленных на ребре прошедшими по нему ранее муравьями. Естественно, что в отличие от весов ребер, которые являются константными, следы феромонов обновляются на каждой итерации алгоритма: как и в природе, со временем следы испаряются, а проходящие муравьи, напротив, усиливают их[4].

Пусть муравей находится в узле , а узел – это один из узлов, доступных для перехода: . Обозначим вес ребра, соединяющего узлы и , как , а интенсивность феромона на нем – как . Тогда вероятность перехода муравья из в будет равна:

$$p_{i,j} = \frac{(\tau_{i,j}^{\alpha})(\eta_{i,j}^{\beta})}{\sum(\tau_{i,j}^{\alpha})(\eta_{i,j}^{\beta})} \tag{1}$$

где

$\tau_{i,j}$ – расстояние от города i до j;

$\eta_{i,j}$ – количество феромонов на ребре ij;

α – параметр влияния длины пути;

β – параметр влияния феромона.

Очевидно, что при $\beta = 0$ алгоритм превращается в классический жадный алгоритм, а при $\alpha = 0$ он быстро сойдется к некоторому субоптимальному решению. Выбор правильного соотношения параметров является предметом исследований, и в общем случае производится на основании опыта.

После того, как муравей успешно проходит маршрут, он оставляет на всех пройденных ребрах след, обратно пропорциональный длине пройденного пути:

$$\tau_{i,j} = (1 - \rho)\tau_{i,j} + \Delta\tau_{i,j}, \tag{2}$$

где

$\rho_{i,j}$ – доля феромона, который испарится;

$\tau_{i,j}$ – количество феромона на дуге ij;

$\Delta\tau_{i,j}$ – количество отложенного феромона.

Количество отложенного феромона, обычно определяется как:

$$\Delta\tau_{i,j}^{k} = \begin{cases} Q/L_k & \text{Если k-ый мурваей прошел по ребру ij;} \\ 0 & \text{Иначе} \end{cases} \tag{3}$$

где

Q – количество феромона, переносимого муравьем;

L_k – стоимость k-го пути муравья (обычно длина).

Трудоёмкость муравьиного алгоритма:

$$O(t * m * n^2) \tag{4}$$

где

t – количество поколений;

m – количество муравьев;

n – количество городов.

Ниже приведены вариации муравьиного алгоритма.

1. **Элитарная муравьиная система**. Из общего числа муравьёв выделяются так называемые «элитные муравьи». По результатам каждой итерации алгоритма производится усиление лучших маршрутов путём прохода по данным маршрутам элитных муравьёв и, таким образом, увеличение количества феромона на данных маршрутах. В такой системе количество элитных муравьёв является дополнительным параметром, требующим определения. Так, для слишком большого числа элитных муравьёв алгоритм может «застрять» на локальных экстремумах.

2. **Max-Min муравьиная система**. Добавляются граничные условия на количество феромонов (max,min). Феромоны откладываются только на глобально лучших или лучших в итерации путях. Все рёбра инициализируются значением max.

3. **Ранговая муравьиная система(ASrank)**. Все решения ранжируются по степени их пригодности. Количество откладываемых феромонов для каждого решения взвешено так, что более подходящие решения получают больше феромонов, чем менее подходящие.

4. **Длительная ортогональная колония муравьёв (COAC)**. Механизм отложения феромонов COAC позволяет муравьям искать реше-

ния совместно и эффективно. Используя ортогональный метод, муравьи в выполнимой области могут исследовать их выбранные области быстро и эффективно, с расширенной способностью глобального поиска и точностью.

2 Конструкторская часть

В данном разделе будут описаны принципы работы выбранных решений и их блоксхемы.

2.1 Разработка реализаций алгоритмов

Основное действие происходит в функции run (рис. 3, 4, 5). На вход ей поступают следующие параметры:

1) time - количество поколений муравьев;
2) ants amount - количество муравьев;
3) elite_one - каждый i-ый(i = elite one) муравей будет элитным;
4) cities - количество городов;
5) eva(evaporation) - процент распыления феромона;
6) al(alpha) - параметр влияния расстояния;
7) be(beta) - параметр влияния феромона;
8) city_from - город появления муравьев;
9) Q, EQ - величина феромона обычного и элитного муравьев.

Они проверяются на корректность в функции run_check(рис. 6). Затем создаются и инициализируются матрицы дистанций, феромонов, массивы муравьев и начинается главный цикл функции - по поколениям. На этом этапе массив муравьев обновляет значения своих элементов до исходных значений(очищаются списки пройденных городов и пройденное расстояние). Затем начинаем обход муравьев. Сначала добавим родной город в список уже посещенных муравьем, чтобы он не мог 'случайно' заскочить домой, не обойдя все остальные города. Дальше начинается третий(по вложенности) цикл. Пока муравей не обошёл все города(can_move() - рис. 7), пусть двигается(make_move - 8). После того, как муравей пройдет все города(выход из цикла), вручную добавляем ему расстояние от текущего города(последнего, в котором он оказался) до родного(города рождения) - иначе получается, что домой он так и не попал, что весьма грустно. Также добавляем его родной город в список посещенных(во второй раз!). Это делается для того, чтобы если мы захотим распечатать путь движения муравья, было видно, что он из дома вышел и домой же вернулся. Дальше 'распыляем' феромоны с помощью функции increase_pheromone - рис. 17. После чего сравниваем лучшее расстояние, что у нас было(хранится в переменной best_way) с пройденным расстоянием муравья и с 0(По умолчанию best_way равен -1). Если одно из этих условий выполняется, обновляем best_way и tabu_list(копию муравьиного массива пройденных городов). Иначе просто идём дальше. И последнее наше действие - уменьшение феромонов на карте. Условно считаем, что каждый следующий муравей начинает движение, когда вернулся прошлый, то есть одновременно на карте присутствует только 1 муравей. Тогда не выходя из цикла по муравьям, сразу после проверки на кратчайший путь уменьшим феромоны по всей карте с помощью функции reduce_pheromone() - рис. 16.

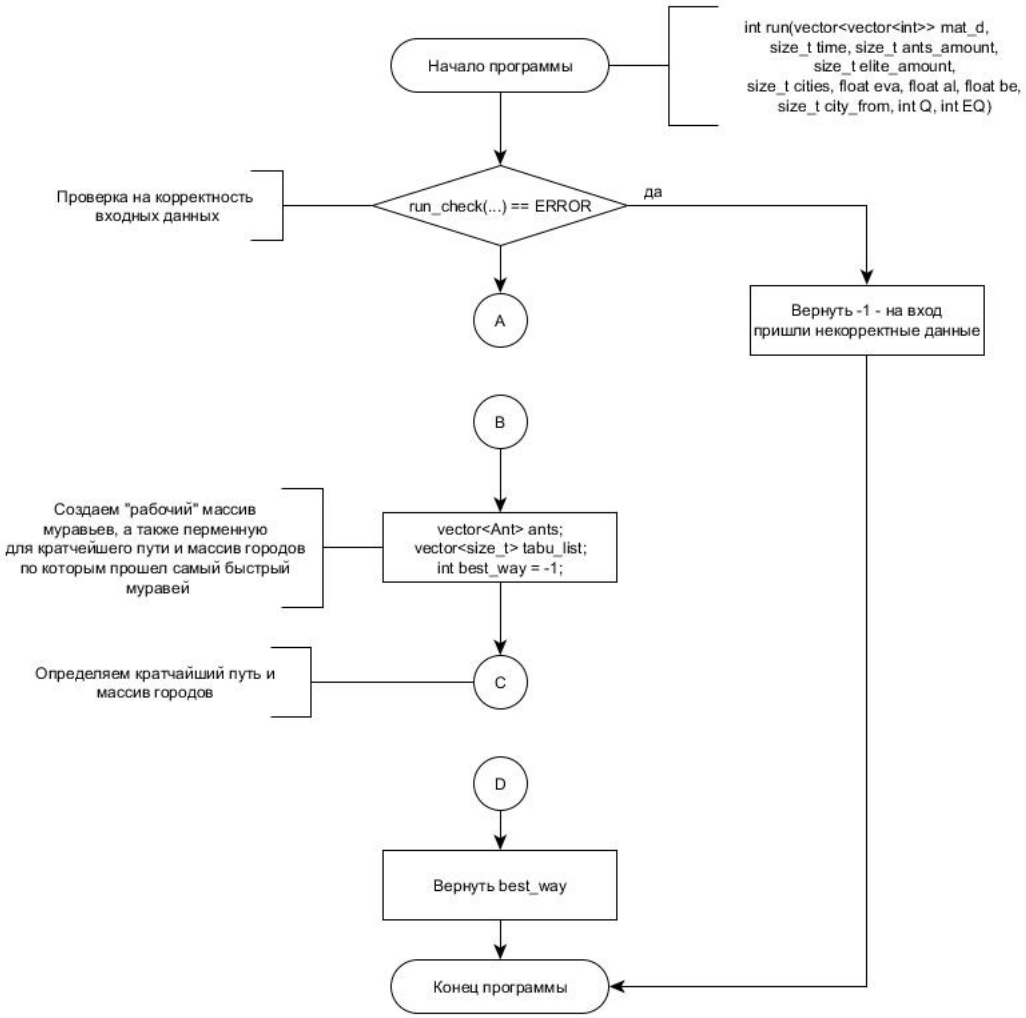

Рис. 3: Основная часть функции run.

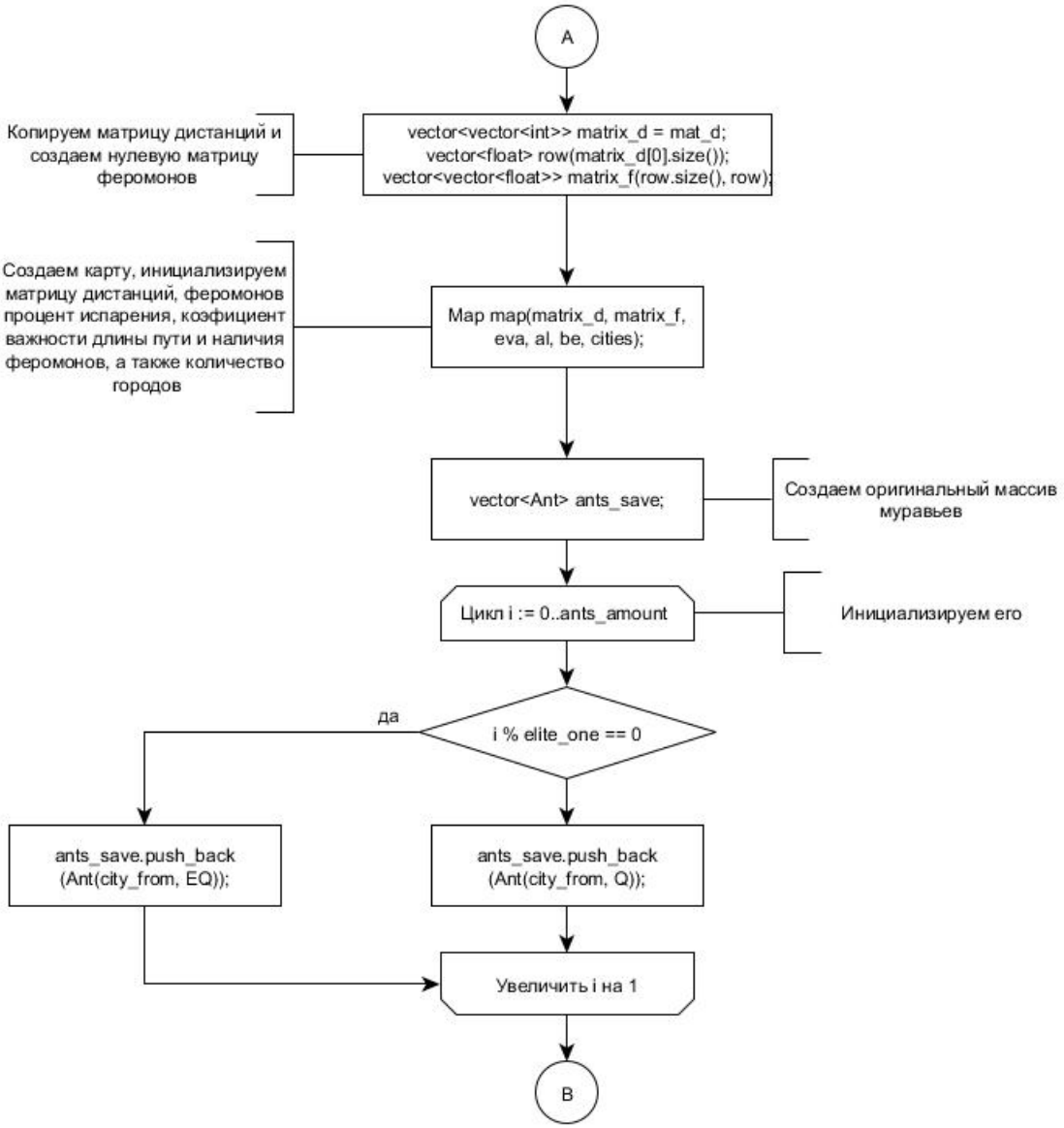

Рис. 4: Инициализация муравьев и матриц.

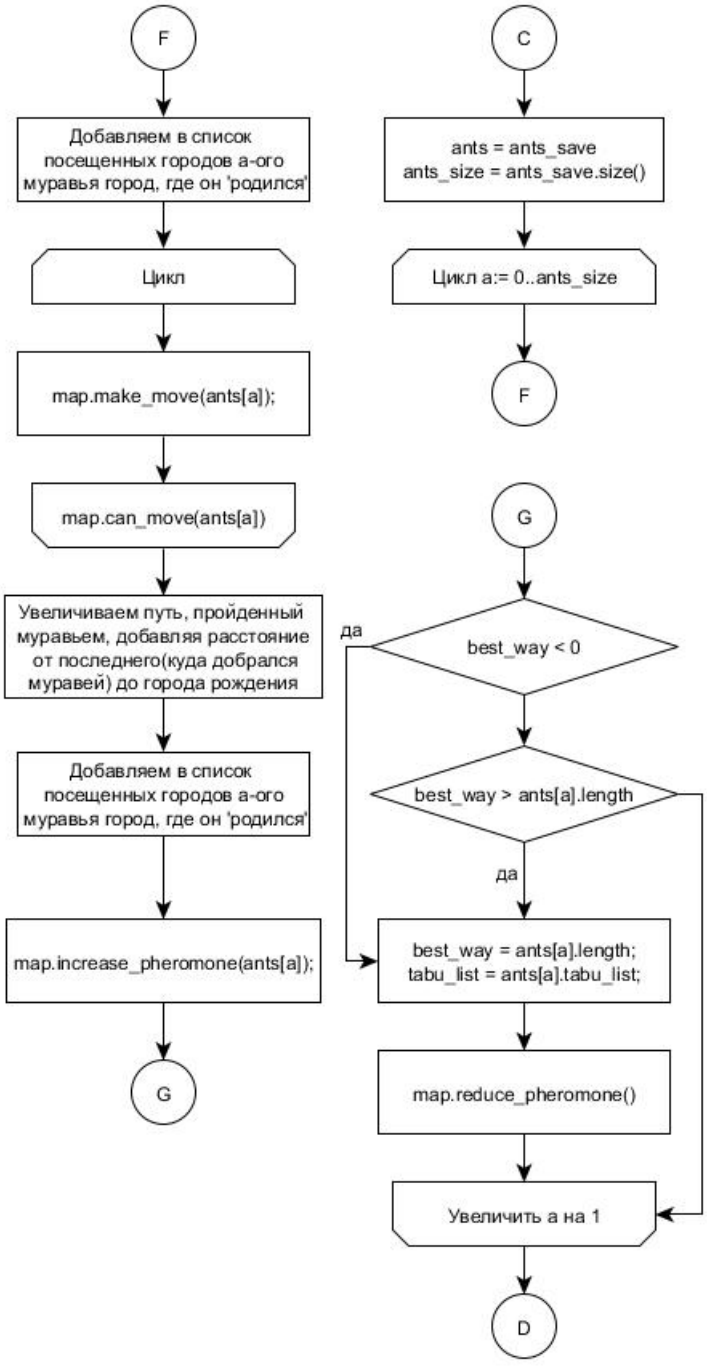

Рис. 5: Процесс поиска кратчайшего пути.

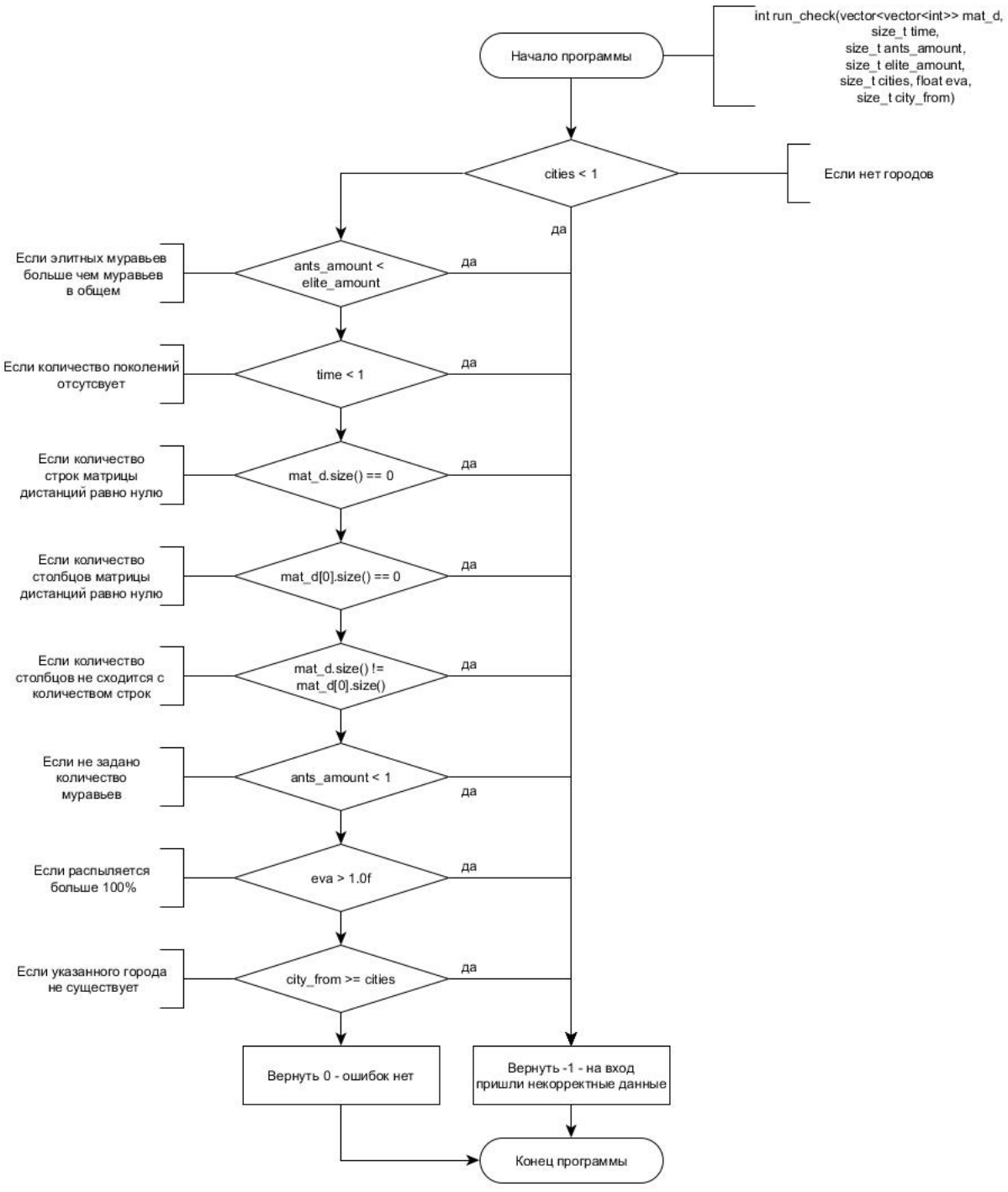

Рис. 6: Проверка корректности входных данных.

Функция can_move(рис. 7) принимает на вход муравья(объект класса Ant - с этого места и дальше просто муравей) и возвращает false(и следовательно прекращает поход муравья), когда размер массива посещенных городов данного муравья становится больше или равен количеству существующих городов. Иначе вернет true и муравей продолжит двигаться.

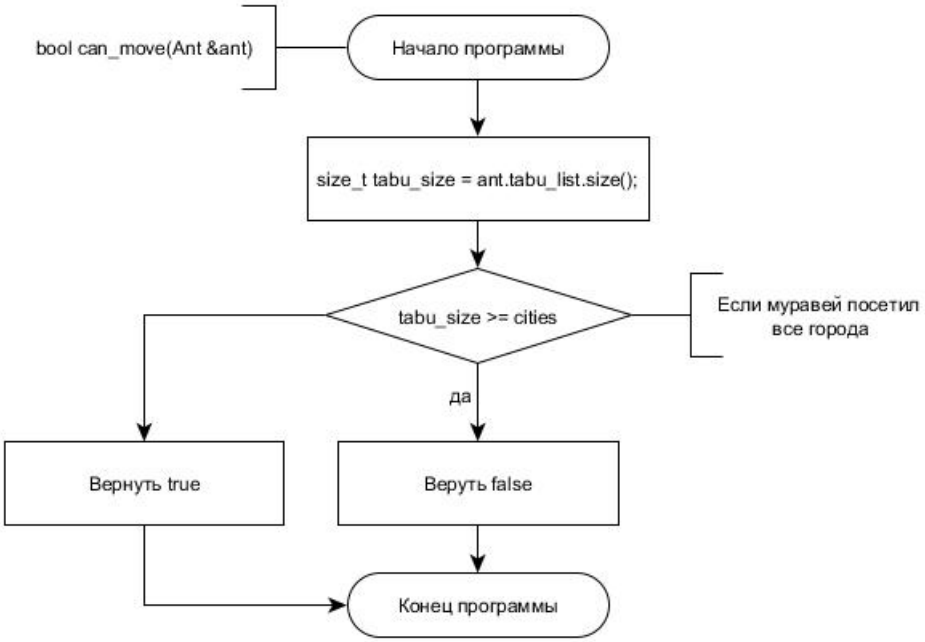

Рис. 7: Функция can_move.

Функция make_move(рис. 8) класса Map принимает на вход муравья и ничего не возвращает. Вначале выполняется поиск(choose_city() - рис. 10) следующего города, куда отправится муравью. После чего определяется расстояние до выбранного города(по матрице дистанций) и дальше функция отдает управление самому муравью, а именно его(класса Ant) одноимённой функции make_move.

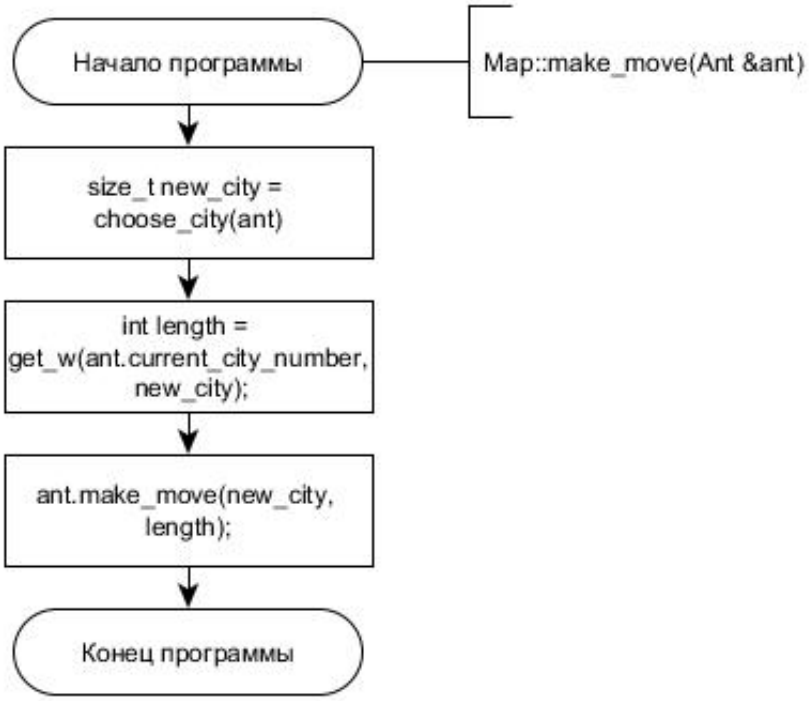

Рис. 8: Функция make_move класса Map.

Муравьиная функция make_move(рис. 9) принимает на вход два параметра: следующий город и расстояние до него. Как и своя тёзка - ничего не возвращает. Эта функция последовательно выполняет следующие 3 действия:

1) помещает номер указанного города в массив посещённых городов;

2) обновляет город пребывания муравья;

3) увеличивает пройденное расстояние муравья на указанное значение.

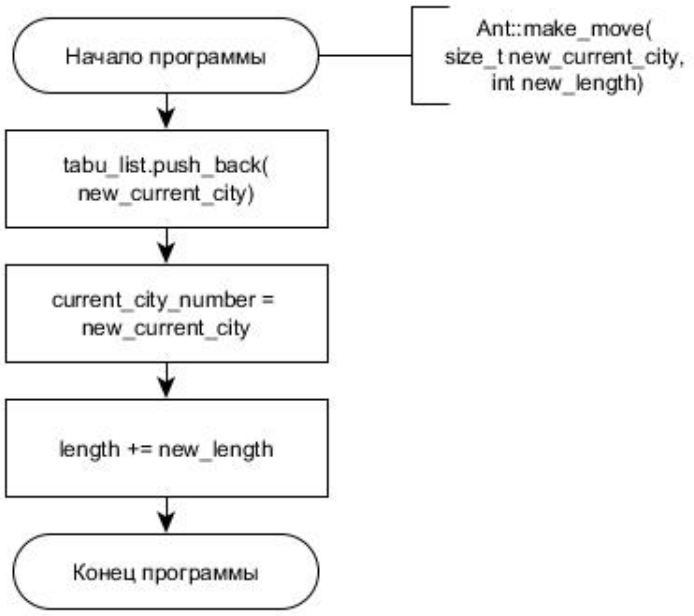

Рис. 9: Функция make_move класса Ant.

Функция choose_city(рис. 10), упомянутая в make_move для Map принимает на вход муравья, а возвращает номер города, который тот 'выбрал' для следующего посещения. Для этого функция получает массив вероятностей(элеемнтом этого массива является пара вида: номер города, вероятность перехода(процент)) выбора каждого из городов. После чего с помощью генератора случайных чисел создаётся число от 0 до 100 и определяется, какому городу принадлежит отрезок, в котором находится данное число. Приведу пример, пусть вероятности попадания в города 1,2,3 равны соответсвенно 30, 60 и 10. Тогда отрезок с 0 по 30 'принадлежит' первому городу, с 31 по 90 - второму, с 90 по 100 - третьму. Например, если число, созданное генератором, равно 95, то муравей пойдкт третий город. Если число равно 40, то пойдет во второй город и т.д. Массив процентов определяется в функции get_p() - рис. 11, 12, 13, 14, о которой пойдет речь дальше.

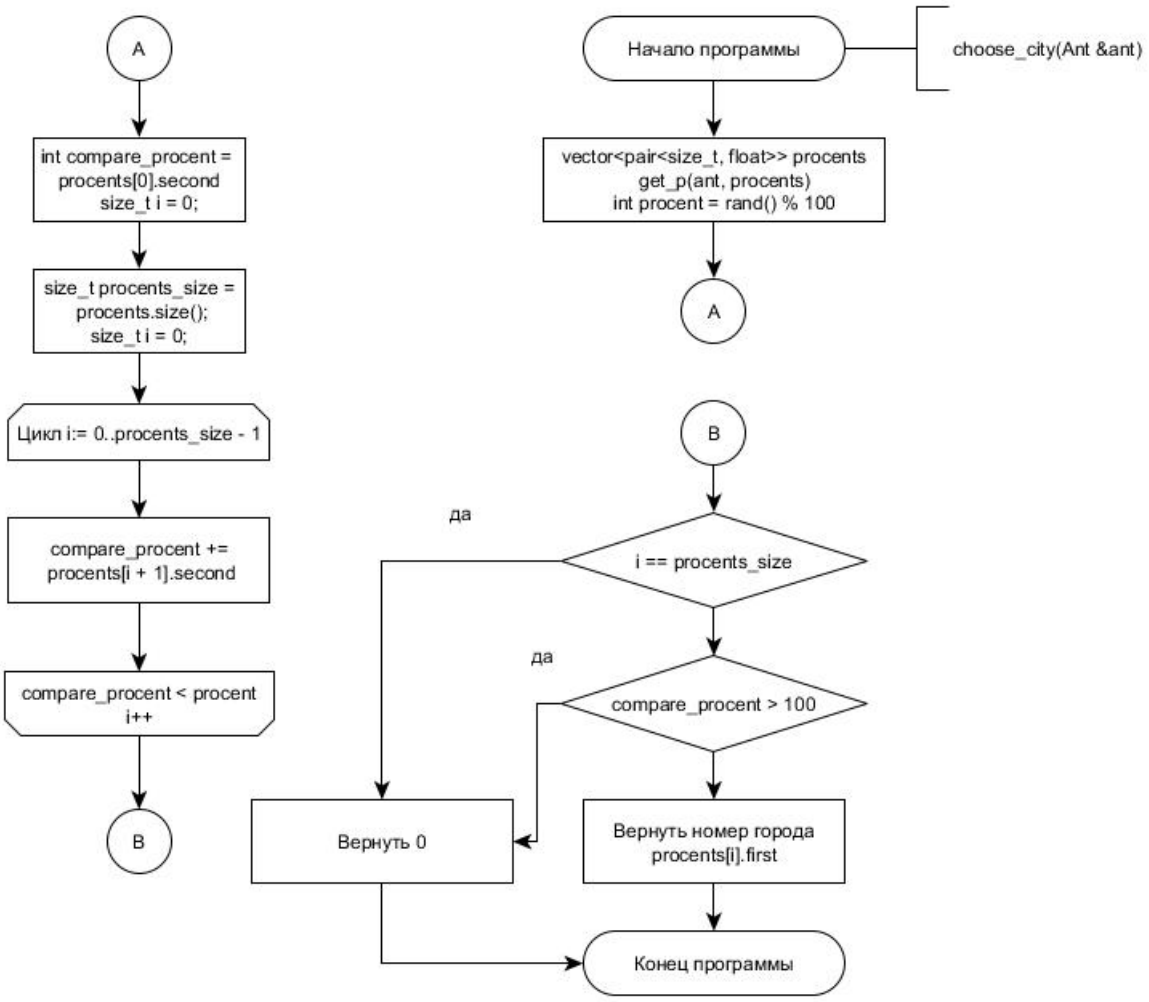

Рис. 10: Функция choose_city.

Функция get_p(рис. 11, 12, 14, 13) получает на вход муравья и ссылку на массив вероятностей. Результатом работы будет обновление этого массива. Первым делом массив вероятностей обнуляется. После чего начинается процесс его заполнения. Проходим по всем городам и проверяем, был ли указанный(пришедший на вход) муравей в i-ом городе. Если был, переключаемся на следующий город, если нет, то вычисляем вероятность похода в этот город. Но перед этим уточняем, что i-ый город не является городом пребывания муравья, для этого достаточно проверить, что путь из текущего города в i-ый не равен нулю. Вероятность определяется по формуле 1. Числитель этой формулы вынесен в отдельную функцию get_p_chisl(рис. 15). По определению вероятность - это отношение исходов некоторого события к общему количеству исходов. На данный момент у нас есть только числитель формулы 1, т. е. мы посчитали на самом деле количество исходов события перехода в i-ый город. Не хватает общего количества исходов. Для того, чтобы посчитать это количество, определим за пределами цикла переменную summ и при каждом вычислении кол-ва исходов для определенного города будем прибавлять это значение к summ. При этом будем сохранять кол-во исходов для определенного города в массив вероятностей procents. После того, как мы обойдем все города, на всякий случай, проверим, что наша переменная summ не ноль(чтобы избежать деления на ноль), а дальше пройдемся по массиву procents и поделим каждое значение на summ. Готово, проценты посчитаны, блоксхемы ниже.

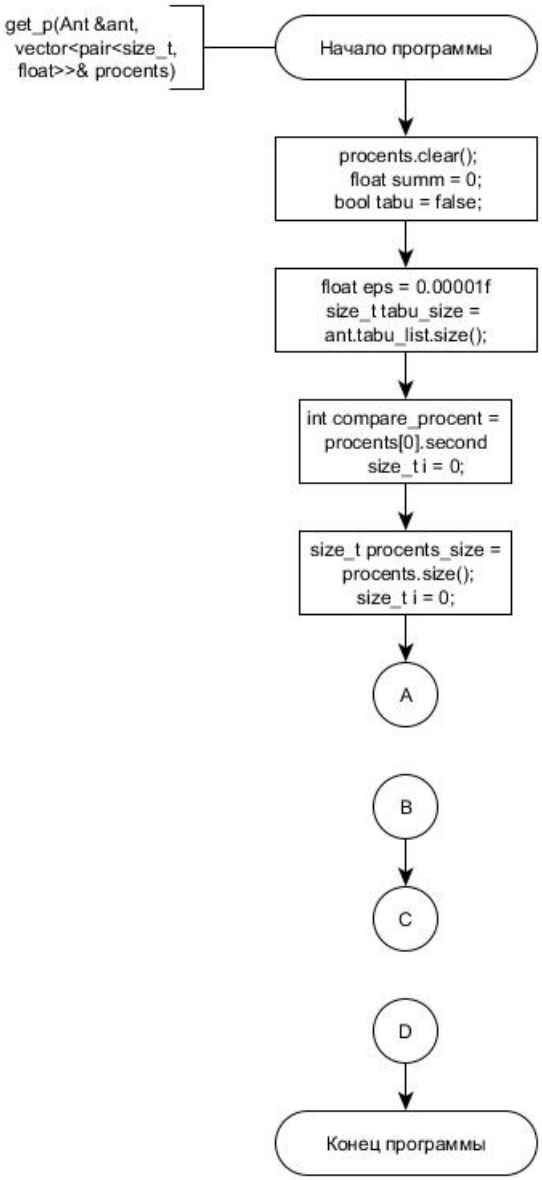

Рис. 11: Основная часть функции get_p.

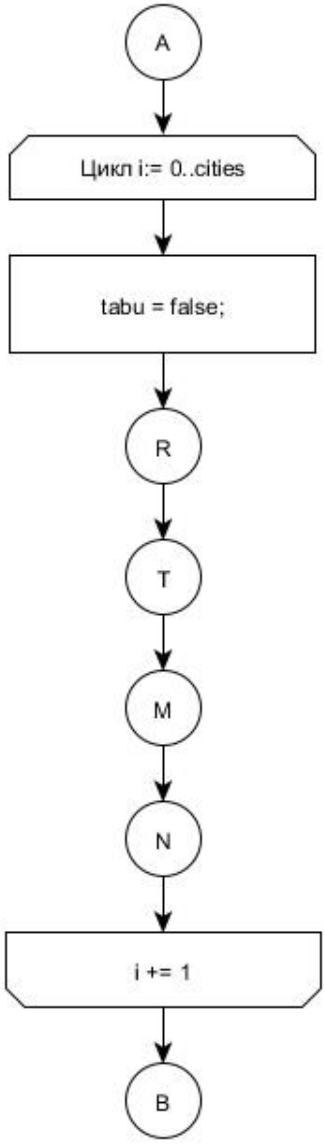

Рис. 12: Цикл по всем городам

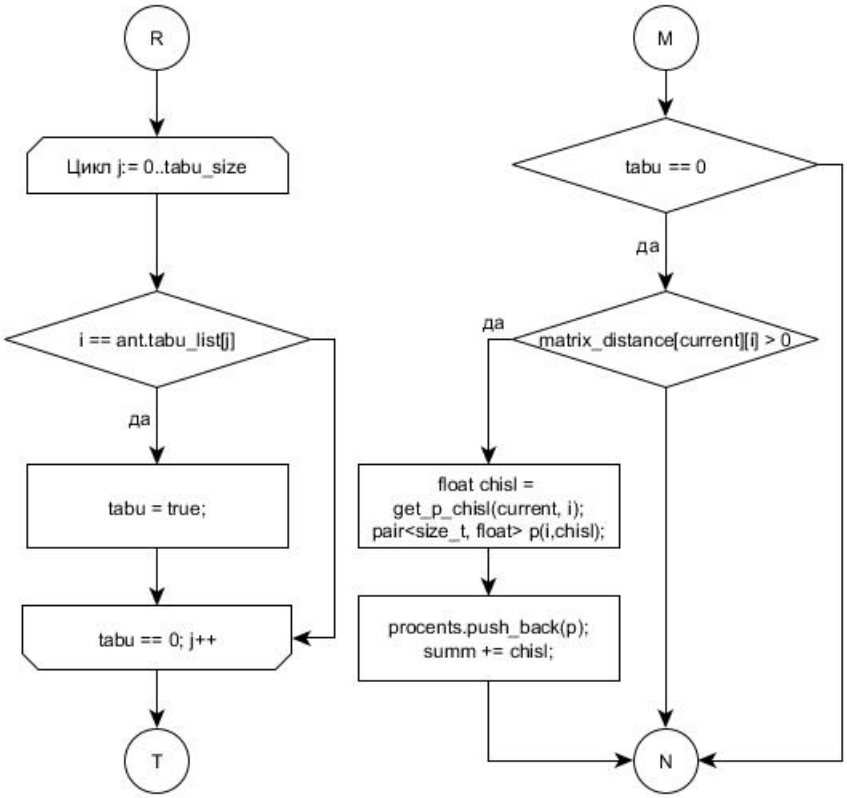

Рис. 13: Определение является ли город посещенным и занесение в массив процентов числителей вероятностей.

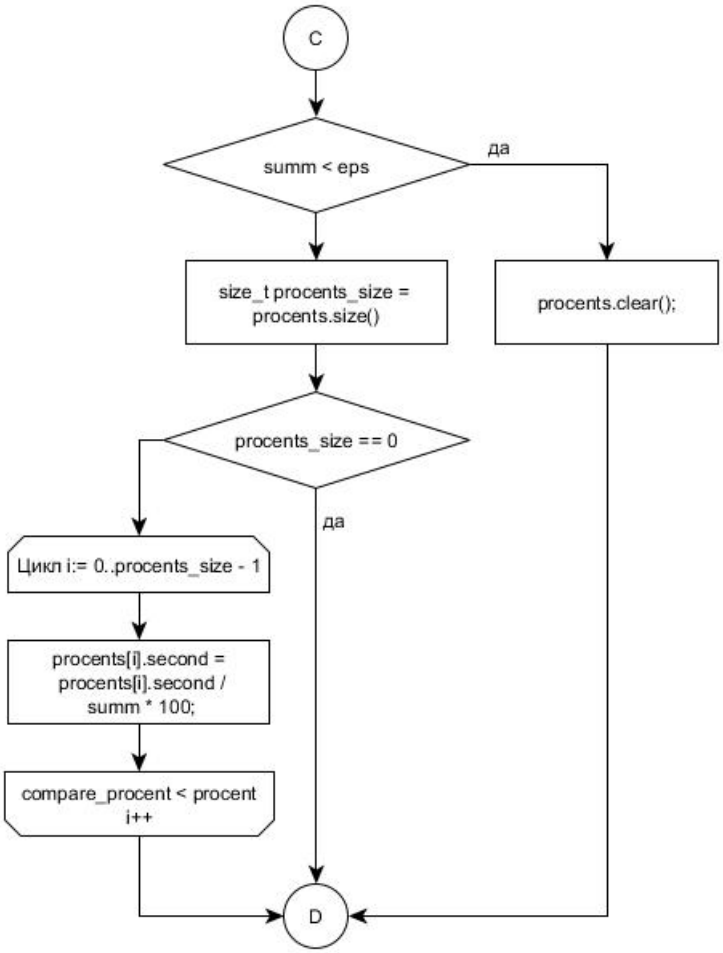

Рис. 14: Деление числа исходов событий на общее количество.

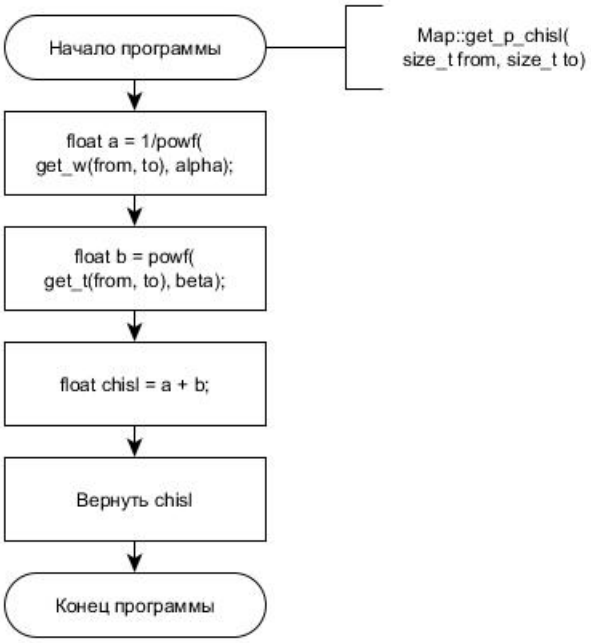

Рис. 15: Программная реализация числителя формулы 1.

Нерассмотренными остались две функции - reduce_pheromone(рис. 16) и increase_pheromone(рис. 17). Кратко о каждой из них. Первая уменьшает количество феромона. Обходим матрицу феромонов и согласно формуле 2 каждое значение умножаем на процент оставшегося феромона = (1 - коэффициент испарения). Однако по формуле 2 к каждому значению надо еще прибавить некоторое число, но я решил вынести это действие в отдельную функцию, а именно в increase_pheromone. Эта функция получает на вход муравья и проходит по массиву пройденных им городов. На каждом шаге выделяет 2 города: прошлый(p) и текущий(c). После чего обновляет ячейку [p][c] и [c][p] в матрице феромонов согласно первому случаю из формулы 3 - прибавляет к ячейке отношение муравьиного феромена к пройденному расстоянию. Таким образом это компенсирует недостающее слагаемое из формулы 2.

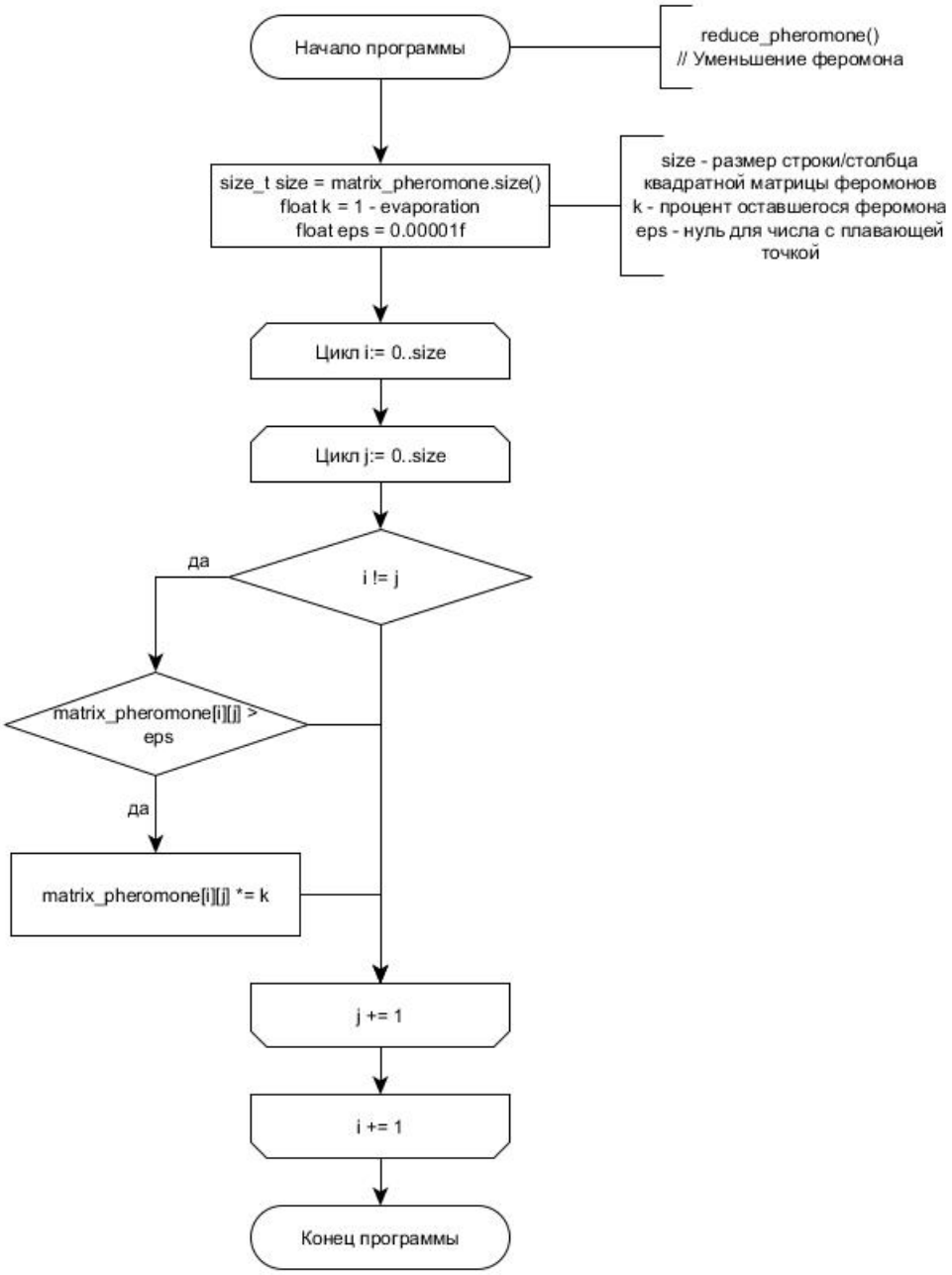

Рис. 16: Функция reduce_pheromone.

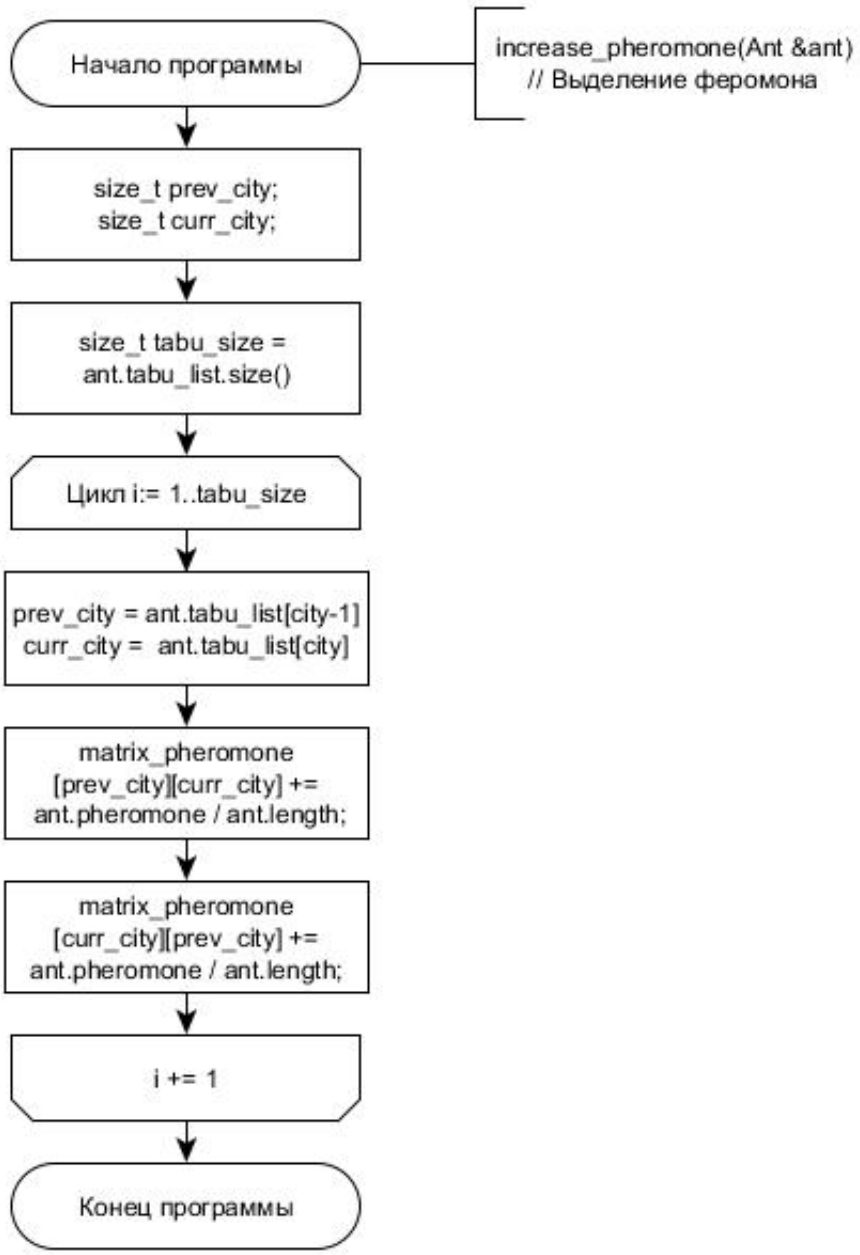

Рис. 17: Функция increase_pheromone.

3 Технологическая часть

В данном разделе будут определены средства реализации и приведен листинг кода.

3.1 Средства реализации

Для реализации программ мною был выбрал язык программирования - C++, поскольку имею опыт работы с ним. Среда разработки - Qt. Время работы процессора замерялось с помощью функции, продемонстрированной в листинге 1. Эта функция в отличие от встроенной функции таймера, способна считать реальное процессорное время работы программы в тиках [5]. Для ее работы была подключена библиотека time.h.

Листинг 1: Функция замера процессороного времени

```
1 unsigned long long tick(void)
2 {
3    unsigned long long d;
4    __asm__ __volatile__ ("rdtsc" : "=A"     (d));
5    return d;
6 }
7
```

3.2 Реализация алгоритмов

Для реализации описанных в конструкторской части решений, было реализовано два класса: Map(листинг 2) и Ant(листинг 3).

Листинг 2: Класс Map

```
class Map {
public:
    Map(vector<vector<int>> matrix_d,
        vector<vector<float>> matrix_p,
        float eva, float al, float be, size_t ci) :
    matrix_distance(matrix_d), matrix_pheromone(matrix_p),
    evaporation(eva), alpha(al), beta(be), cities(ci){;}

    // Матрица дистанций и феромонов
    vector<vector<int>> matrix_distance;
    vector<vector<float>> matrix_pheromone;

    // Интенсивность испарения. От 0 до 1.
    float evaporation;
    // Важность веса ребра и уровня феромонов соответственно
    float alpha, beta;
    // Количество городов
    size_t cities;

    // Возвращает ложь, если пройдены все города, иначе истину
    bool can_move(Ant &ant) {
        size_t tabu_size = ant.tabu_list.size();
        if (tabu_size >= this->cities) {
            return false;
        }
        return true;
    }

    // Уменьшение феромона
    void reduce_pheromone();

    // Выделение нового феромона
    void increase_pheromone(Ant &ant);

    // Обновить данные муравья
    void make_move(Ant &ant);

private:
    // Получить массив исходов
    float get_p_chisl(size_t from, size_t to);

    // Получить вес ребра для перехода. Если <0 значит ребра нет
    int get_w(size_t from, size_t to);

    // Получить интенсивность феромона
    float get_t(size_t from, size_t to);

    // Выбор следующего города
```

```
49      size_t choose_city(Ant &ant);
50
51      // Получить массив вероятностей походов
52      void get_p(Ant &ant, vector<pair<size_t, float>>& procents);
53  };
54
```

Листинг 3: Класс Ant

```
1   class Ant {
2   public:
3       Ant(size_t ccn, int Q) {
4         current_city_number = ccn;
5         length = 0;
6         pheromone = Q;
7       }
8       vector<size_t> tabu_list; // массив посещенных городов
9       size_t current_city_number; // номер текущего города
10      int length; // пройденное расстояние
11      float pheromone; // выделение феромона
12
13      // Поместить текущий номер в массив посещенных городов
14      // Установить ноый номер текущего города
15      // Обновить пройденное расстояние
16      void make_move(size_t new_current_city, int new_length);
17  };
18
```

Реализация объявленных функций продемонстрирована в следующих листингах:

листинг 4 - run_check;

листинг 5 - run;

листинг 8 - get_w;

листинг 6 - get_p_chisl;

листинг 7 - get_p;

листинг 9 - get_t;

листинг 10 - choose_city;

листинг 11 - Ant::make_move;

листинг 12 - Map::make_move;

листинг 13 - reduce_pheromone;

листинг 14 - increase_pheromone.

Листинг 4: Функция run_check - проверить входные данные функции run

```
#define NOERROR 0
#define ERROR -1

int run_check(vector<vector<int>> mat_d,
              size_t time, size_t ants_amount,
              size_t elite_amount,  size_t cities, float eva,
              size_t city_from) {

    // Если городов нет
    if (cities < 1) {
        std::cout << "\n run_check() - cities < 1";
        return ERROR;
    }
    // Если элитных муравьев больше чем муравьев в общем
    if (ants_amount < elite_amount) {
        std::cout << "\n run_check() - elite_amount error";
        return ERROR;
    }
    // Если поколений муравьев нет
    if (time < 1) {
        std::cout << "\n run_check() - time < 1";
        return ERROR;
    }
    // Если с матрией расстояний что то не так
    if (mat_d.size() == 0 ||
            mat_d[0].size() == 0 ||
            mat_d.size() != mat_d[0].size()) {
        std::cout << "\n run_check() - mat_d error";
        return ERROR;
    }

    // Если муравьев нет
    if (ants_amount < 1) {
        std::cout << "\n run_check() - ants_amount < 1";
        return ERROR;
    }

    // Если распыляется больше 100%
    if (eva > 1.0f) {
        std::cout << "\n run_check() - eva > 1.0";
        return ERROR;
    }

    // Если указанного города не существует
    if (city_from >= cities) {
        std::cout << "\n run_check() - city_from >= cities";
        return ERROR;
    }
    return NOERROR;
}

```

Листинг 5: Функция run - получить кратчайший путь

```
int run(vector<vector<int>> mat_d, size_t time, size_t ants_amount, size_t
    elite_one,  size_t cities, float eva, float al, float be,  size_t city_from, int Q
    , int EQ) {
    // Проверка на корректность входных данных
    if (run_check(mat_d,time, ants_amount, elite_one, cities,
                  eva, city_from) == ERROR) {
        return ERROR;
    }
    // Создаем матрицу дистанций и феромонов
    vector<vector<int>> matrix_d = mat_d;
    vector<float> row(matrix_d[0].size());
    vector<vector<float>> matrix_f(row.size(), row);

    Map map(matrix_d, matrix_f, eva, al, be, cities);
    vector<Ant> ants_save;

    for (size_t i = 0; i < ants_amount ; i++) {
        if (i % elite_one == 0) {
            ants_save.push_back(Ant(city_from, EQ));
        } else ants_save.push_back(Ant(city_from, Q));
    }
    vector<size_t> tabu_list;
    vector<Ant> ants;
    int best_way = -1;

    size_t ants_size = ants_save.size();
    for (size_t t = 0; t < time; t++) {
        ants = ants_save;
        for (size_t a = 0; a < ants_size; a++) {
            ants[a].tabu_list.push_back(city_from);
            while (map.can_move(ants[a])) map.make_move(ants[a]);

            // Выполняем дополнительный шаг, возвращающий муравья в начало
            if (ants[a].tabu_list.back() < map.matrix_distance.size()) {
                ants[a].length += map.matrix_distance[ants[a].tabu_list.back()][
    city_from];
                ants[a].tabu_list.push_back(city_from);
            } else std::cout << "\n strange error";

            map.increase_pheromone(ants[a]);

            if (best_way < 0 || best_way > ants[a].length) {
                best_way = ants[a].length;
                tabu_list = ants[a].tabu_list;
            }
            map.reduce_pheromone();
        }
    }
    return best_way;
}

```

Листинг 6: Функция get_p_chisl - получить числитель формулы 1

```
float Map::get_p_chisl(size_t from, size_t to) {
    float a = 1/powf(get_w(from, to), alpha);
    float b = powf(get_t(from, to), beta);
    float chisl = a + b;
    return chisl;
}

```

Листинг 7: Функция get_p - получить массив вероятностей

```
   void Map::get_p(Ant &ant, vector<pair<size_t, float>>& procents) {
    procents.clear();
    float summ = 0;
    bool tabu = false;

    float eps = 0.00000001f;
    size_t tabu_size = ant.tabu_list.size();
    // Цикл по всем городам
    for(size_t i = 0; i < cities; i++){
       tabu = false;
       for (size_t j = 0; j < tabu_size && !tabu; j++) {
          if (i == ant.tabu_list[j]) {
             // Если был, то пометим это
             tabu = true;
          }
       }
       if (!tabu) {
          size_t current = ant.current_city_number;
          if (get_w(current, i) > 0) {
             float chisl = get_p_chisl(current, i);
             pair<size_t, float> p(i,chisl);
             procents.push_back(p);
             summ += chisl;
          }
       }
    }
    if (summ < eps) {
       procents.clear();
       std::cout << "\nzero division";
       return;
    }
    size_t procents_size = procents.size();
    if (procents_size == 0) {
       std::cout << "\nError with adding elements";
       return;
    }
    for (size_t i = 0; i < procents_size; i++) {
       procents[i].second = procents[i].second / summ * 100;
    }
}

```

Листинг 8: Функция get_w - получить расстояние до города to из from

```
int Map::get_w(size_t from, size_t to) {
    return matrix_distance[from][to];
}

```

Листинг 9: Функци я get_t - получить количество феромонов на пути к городу to из города from

```
float Map::get_t(size_t from, size_t to) {
    return matrix_pheromone[from][to];
}

```

Листинг 10: Функция choose_city - Выбрать следующий город для муравья

```
    size_t Map::choose_city(Ant &ant) {
    vector<pair<size_t, float>> procents;
    get_p(ant, procents);
    int procent = rand() % 100;
    int compare_procent = procents[0].second;

    size_t procents_size = procents.size();
    size_t i = 0;
    for (;i < procents_size - 1 && compare_procent < procent;i++){
      compare_procent += procents[i + 1].second;
    };
    if (i == procents_size) {
      std::cout << "\n choose_city out of borders " <<
                i << "/" << procents_size;
      return 0;
    }

    if (compare_procent > 100) {
       std::cout << "\n choose_city() error procenting " <<
                 compare_procent;
       return 0;
    }
    return procents[i].first;
}

```

Листинг 11: Функция ant::make_move

```
void Ant::make_move(size_t new_current_city, int new_length) {
    tabu_list.push_back(new_current_city);
    current_city_number = new_current_city;
    length += new_length;
}

```

Листинг 12: Функция map::make_move

```
void Map::make_move(Ant &ant) {
    size_t new_city = choose_city(ant);
    int length = get_w(ant.current_city_number, new_city);
    ant.make_move(new_city, length);
}

```

Листинг 13: Функция reduce_pheromone - уменьшает количество феромонов на всех узлах матрицы феромонов

```
        void Map::reduce_pheromone() {
    // Матрица квадратная поэтому достаточно одной величины
    size_t size = matrix_pheromone.size();
    float k = 1 - evaporation;
    float eps = 0.00001f;

    for (size_t i = 0; i < size; i++) {
        for (size_t j = 0; j < size; j++) {
            if (i != j && matrix_pheromone[i][j] > eps) {
                matrix_pheromone[i][j] *= k;
            }
        }
    }
}

```

Листинг 14: Функция increase_pheromone - увеличивает количество феромонов для пройденных муравьем городов

```
    void Map::increase_pheromone(Ant &ant) {
    size_t prev_city;
    size_t curr_city;
    size_t tabu_size = ant.tabu_list.size();

    for (size_t city = 1; city < tabu_size; city++) {
        prev_city = ant.tabu_list[city-1];
        curr_city =  ant.tabu_list[city]; matrix_pheromone[prev_city][curr_city];
        matrix_pheromone[prev_city][curr_city] += ant.pheromone / ant.length;
        matrix_pheromone[curr_city][prev_city] += ant.pheromone / ant.length;
    matrix_pheromone[prev_city][curr_city];
    }
}

```

4 Экспериментальная часть

В данном разделе будет сравнительный анализ муравьиного алгоритма в зависимости от выбранных параметров на основе экспериментальных данных.

4.1 Сравнительный анализ

Для экспериментов использовалась матрица расстояний 10x10, изображенная в таблице 1. Количество повторов каждого эксперимента = 50. Результат одного эксперимента рассчитывается как средний из результатов проведенных испытаний с одинаковыми входными данными.

Таблица 1: Матрица расстояний между 10 городами.

0	4	5	7	8	9	2	3	4	10
4	0	3	2	6	7	2	5	9	3
5	3	0	9	8	7	6	7	8	9
7	2	9	0	4	1	8	5	6	6
8	6	8	4	0	7	4	2	1	8
9	7	7	1	7	0	8	3	2	1
2	2	6	8	4	8	0	9	9	9
3	5	7	5	2	3	9	0	9	9
4	9	8	6	1	2	9	9	0	9
10	3	9	6	8	1	9	9	9	0

Было проведено несколько опытов. Первый проводился с 50 муравьями, коэффициентом распыления 0.1, без элитных муравьев и в 200 поколений. Замерялось, как влияют коэффициенты "жадности"(α) и "стадности"(β). По умолчанию $\alpha = 0$, а $\beta = 1$. Затем α увеличивается на 0.05 до 1, а β понижается до 0 с тем же шагом 0.05. Результаты занесены в таблицу 2. Она состоит из 4-ёх столбцов: корректность(не при всех коэффициентах получался правильный ответ), α , β и скорость работы. Как видно из рисунков 18 и 19 наиболее быстрым и в то же время правильным решением является использование коэффициентов $\alpha = 0.7$, $\beta = 0.3$, хотя разница по сравнению с другими значениями не столь велика и при других значениях прочих коэффициентов - количества обычных и элитных муравьев или поколений, а также коэффициента испарения - мы можем получить другие результаты.

Таблица 2: Зависимость скорости и корректности решения от коэффициентов α и β.

Корректность (от 0 до 1)	α (от 0 до 1)	β (от 0 до 1)	скорость работы (в тиках)
0.93	0.00	1.00	552793632
0.90	0.05	0.95	671431366
0.93	0.10	0.90	677701845
1.00	0.15	0.85	688931058
1.00	0.20	0.80	672681069
1.00	0.25	0.75	665585706
0.93	0.30	0.70	678160444
1.00	0.35	0.65	685878101
1.00	0.40	0.60	772171866
0.97	0.45	0.55	650128055
0.97	0.50	0.50	641031277
1.00	0.55	0.45	664468673
1.00	0.60	0.40	643548411
1.00	0.65	0.35	645030182
1.00	0.70	0.30	641664003
1.00	0.75	0.25	647894697
1.00	0.80	0.20	654792064
1.00	0.85	0.15	670107572
0.93	0.90	0.10	675516409
1.00	0.95	0.05	663381617
1.00	1.00	0.00	718245238

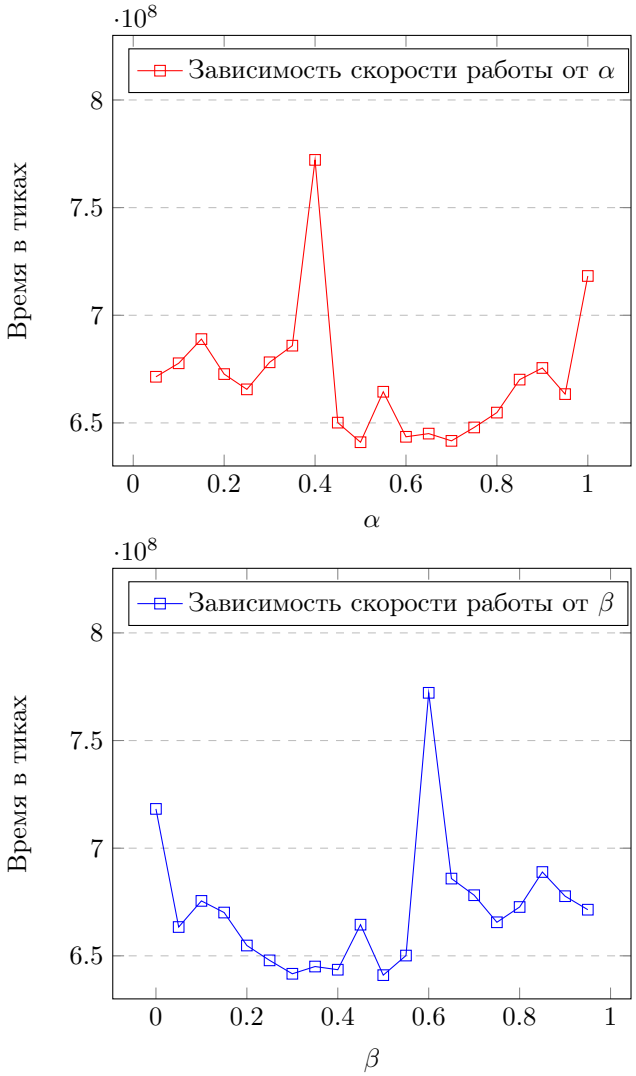

Рис. 18: График зависимости скорости работы от α

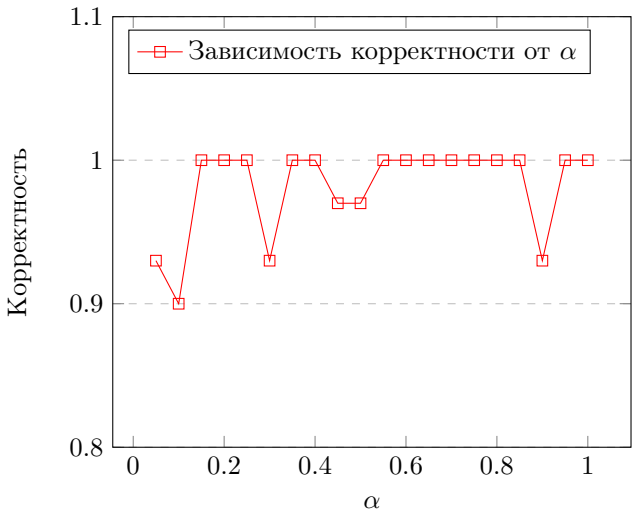

Рис. 19: График зависимости корректности ответа от α.

Теперь проанализируем зависимость от количества муравьев и количества поколений. α и β теперь будут константами,а именно $\alpha = 5$ и $\beta = 12$. Количество муравьев варьируется от 10 до 80 с шагом сначала 10, потом 20, 40 и т.д., количество поколений от 20 до 650 с шагом сначала 10, потом 20, 40 и т.д. Результаты замеров сведены в таблицу 3. Она также состоит из 4 столбцов. Первый столбец вновь означает корректность, а четвертый - время работы. А второй и третий - 'количество муравьев' и 'количество замеров' соответственно. Рисунок 20 подтверждает формулу 4 расчёта трудоёмкости муравьиного алгоритма. Более интересен рисунок 21, который показывает, что при 20-и поколениях нет ни одного правильного ответа, при поколениях от 30 до 50 правильные ответы появляются только если взять более 40 муравьев. От 90 до 330 правильный ответ выводится уже при 20 муравьях, а если поколений 650, то и при 10 муравьях ответ будет правильный. Напомню, что обход происходит по 10 городам.

Таблица 3: Зависимость скорости и корректности решения от количества муравьев и поколений.

Корректность (от 0 до 1)	Количество муравьев (в штуках)	Количество поколений (в штуках)	скорость работы (в тиках)
0.93	10	20	11123431
0.93	20	20	23200249
0.93	40	20	45064059
0.93	80	20	89106562
0.93	10	30	15277990
0.93	20	30	30652063
1.00	40	30	62648153
1.00	80	30	132719349
0.93	10	50	27005527
0.93	20	50	52959829
1.00	40	50	109892963
1.00	80	50	211573758
0.93	10	90	47740420
1.00	20	90	94700006
1.00	40	90	199694544
1.00	80	90	387066569
0.93	10	170	90954856
1.00	20	170	181294108
1.00	40	170	363646870
1.00	80	170	729324525
0.93	10	330	174772204
1.00	20	330	344393887
1.00	40	330	712391169
1.00	80	330	1457121092
1.00	10	650	348804549
1.00	20	650	719097981
1.00	40	650	1431310052
1.00	80	650	2878805475

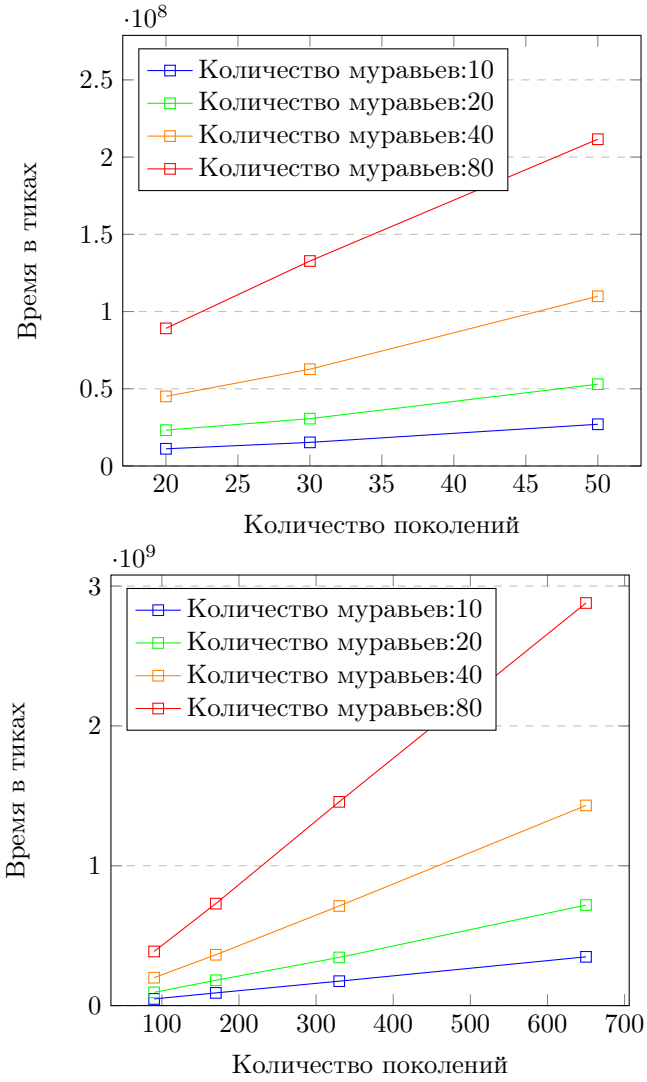

Рис. 20: График зависимости времени работы алгоритма от количества муравьев и поколений.

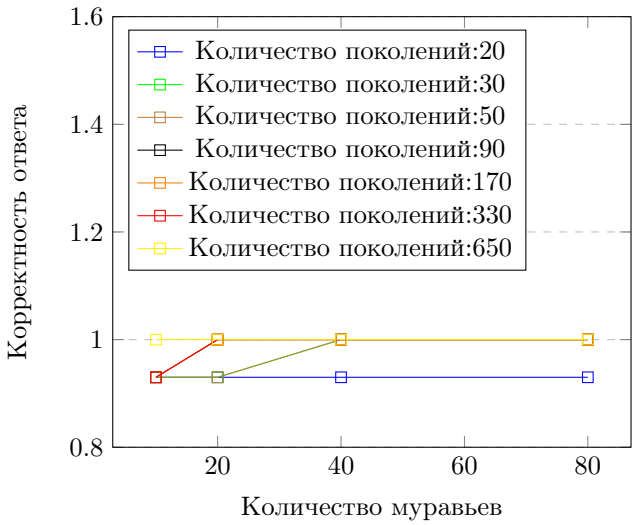

Рис. 21: График зависимости корректности работы алгоритма от количества муравьев и поколений.

Настало время проверить, как сильно изменится скорость и корректность, если начать менять выделяемое муравьем количество феромонов и добавить элитных с повышенным количеством. $\alpha = 5$ и $\beta = 12$. Количество выделяемого феромона у обычного муравья будет варьироваться от 1 до 5 с шагом 2. У элитного всегда 10. Количество элитных изменяется от 50 до 1. Суммарное количество муравьев всегда равно 50. Процент испарения 0.1. Результаты работы приведены в таблице 4, где колчиество феромонов и количество элитных муравьев это 2 и 3 столбец. 1 и 4 как всегда корректность и затраченное время. Если посмотреть на рисунок 22, то можно заметить, что лучшие результаты достигаются, когда обычный муравей выделяет мало феромона(по сравнению с элитными) и при этом среди 50 муравьев 3 или 7 являются элитными(примерно 7 и 14 процентов от общего количества). Когда элитных муравьев слишком много или обычные муравьи выделяют феромона почти как элитные, то все преимущество наличия элитных муравьев исчезает, поскольку все муравьи примерно одинаковы. На рисунке 23 продемонстрирована зависимость корректности ответа от количества элитных муравьев и феромонов.

Таблица 4: Зависимость скорости и корректности решения от количества элитных муравьев и выделяемого феромона обычным муравьем.

Корректность (от 0 до 1)	Количество феромона, выделяемое обычным муравьем(в штуках)	Количество элитных муравьев(в штуках)	скорость работы (в тиках)
1.00	1	1	559773514
1.00	3	1	549665664
1.00	5	1	599716769
1.00	1	3	544924989
1.00	3	3	573101663
0.93	5	3	579858919
1.00	1	7	539835456
1.00	3	7	579842625
0.90	5	7	574067559
0.93	1	16	579406567
0.93	3	16	576663013
0.90	5	16	574714519
0.90	1	50	572369419
0.90	3	50	572640859
1.00	5	50	576265339

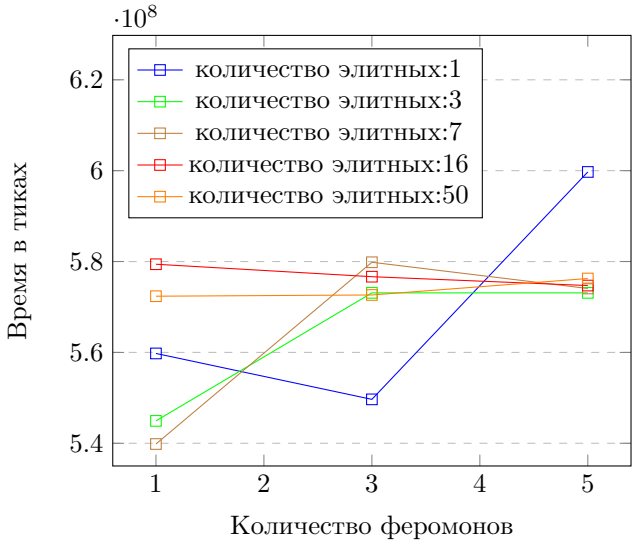

Рис. 22: График зависимости времени работы алгоритма от количества элитынх муравьев и феромонов.

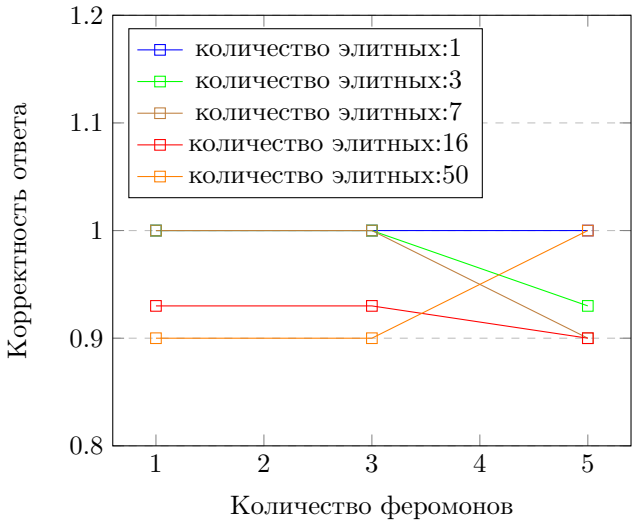

Рис. 23: График зависимости корректности работы от количества элитных муравьев и феромонов обычного муравья.

Теперь измерим влияние процента испарения феромона. Поскольку оно привязано к количеству феромона, будем варьировать его(β) от 2 до 6, при этом $\alpha = 1$. Процент испарения изменяется от 0 до 0.9 с шагом 0.1. Элитных муравьев исключим, количество феромонов, переносимое одним муравьем, установим в единицу. Чтобы увеличить количесво некорректных ответов, зададим малое количество муравьев и поколений - 30 и 20 соответственно. Полученные результаты занесены в таблицу 5, которая состоит из следующих стобцов: корректность, β(можно рассматривать как отношение β к

α), коэффициент испарения, время работы. Любопытно, что из рисунка 24 видно, что наиболее хорошим значением коэффициента испарения является 0.1, но при нем алгоритм работает неправильно, когда β=2. Наиболее общий результат наблюдается при коэффициенте испарения, равным 0.4, все три замера дают почти идентичную скорость работы, причем для β=2 и β=6 наибольшую.

Таблица 5: Зависимость скорости и корректности решения от испарения и влияния количества феромонов

корректность (от 0 до 1)	β	коэффициент испарения (от 0 до 1)	скорость работы (в тиках)
0.93	2	0.00	32618353
1.00	4	0.00	33330212
1.00	6	0.00	32412299
0.93	2	0.10	32272986
1.00	4	0.10	31939012
0.93	6	0.10	32491167
1.00	2	0.20	32825864
1.00	4	0.20	32374316
0.90	6	0.20	31968868
0.93	2	0.30	31947558
1.00	4	0.30	32982935
0.90	6	0.30	32648815
1.00	2	0.40	32345549
1.00	4	0.40	32227624
1.00	6	0.40	32410232
1.00	2	0.50	32434753
0.93	4	0.50	33008954
1.00	6	0.50	32432297
0.93	2	0.60	33403780
0.93	4	0.60	32512934
1.00	6	0.60	32638150
0.93	2	0.70	32806587
1.00	4	0.70	33044899
0.93	6	0.70	33249484
1.00	2	0.80	33009915
0.93	4	0.80	33375704
0.93	6	0.80	33231976
1.00	2	0.90	33298340
0.93	4	0.90	33041315
1.00	6	0.90	33014037

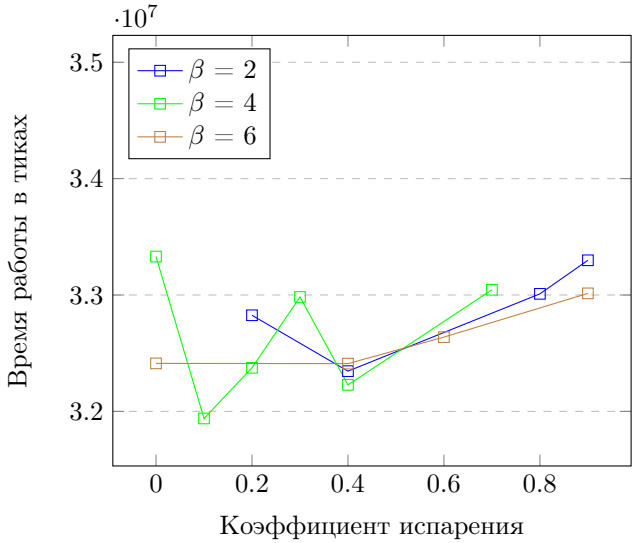

Рис. 24: График зависимости скорости работы от коэффициента испарения с учетом, что ответ верен.

Вывод

Скорость муравьиного алгоритма колеблется в зависимости от выбранных параметров. Наиболее быстрое решение удалось достичь при $\alpha = 0.7$, $\beta = 0.3$. Оптимальное количество поколений для 10 городоа составляет 30, количество муравьев - 40. Введение элитных муравьев ускоряет процесс, однако лишь в том случае когда процент элитных составляет 7-14 процентов от общего числа. Опытным путем было определено, что оптимальным соотношением количества феромонов обычного муравья к количеству феромонов элитного муравья является 0.1. Так, у обычного например 1, а у элитного 10. Коэффициент испарения - 0.4 .

Заключение

В ходе работы был изучаен и реализован оригинальный муравьиный алгоритм, а также его улучшение - введение в строй элитных муравьей. Были получены наиболее оптимальные параметры для быстрого и правильного решения задачи коммивояжера на матрице из десяти городов.

Список литературы

[1] Описание муравьиного алгоритма https://wiki2.org/ru/Муравьиный

[2] Библиография муравьиного алгоритма https://web.archive.org/web/20090204054713/http://lasius.narod.ru/antRef/antDiss/aco-5.html

[3] Пример работы муравьиного алгоритма https://commons.wikimedia.org/wiki/File:Aco$_b$*ranches.svg?uselang* = *ru*

[4] Формулы выбора города, обновления феромонов https://habr.com/post/105302/

[5] И.В. Ломовской. Курс лекций по языку программирования C, 2017