



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

ФАКУЛЬТЕТ

«Информатика и системы управления»

КАФЕДРА

«Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ

По лабораторной работе №6

По курсу: «Анализ алгоритмов»

Тема: «Муравьиный алгоритм»

Студент:

Пронин А. С.

Группа:

ИУ7-52Б

Преподаватель:

Волкова Л. Л.

Оценка:

Москва

2021

Содержание

Введение	3
1 Аналитический раздел	4
1.1 Описание задачи	4
1.2 Описание алгоритма "Рекурсивный полный перебор" . . .	5
1.3 Описание алгоритма "Муравьиный алгоритм"	5
1.4 Вывод	7
2 Конструкторский раздел	8
2.1 Схемы алгоритмов	8
2.2 Вывод	11
3 Технологический раздел	12
3.1 Требование к ПО	12
3.2 Выбор инструментов	12
3.3 Реализация алгоритмов	13
3.4 Тестирование	17
3.5 Вывод	18
4 Исследовательский раздел	19
4.1 Технические характеристики	19
4.2 Примеры работы программы	19
4.3 Постановка экспериментов	20
4.4 Вывод	23
Заключение	24
Список использованных источников	25

Введение

Цель работы – изучить муравьиный алгоритм на материале решения задачи коммивояжёра.

Задачи работы:

- описать методы решения;
- описать реализацию, реализовать метод;
- выбрать класс данных, составить набор данных;
- провести параметризацию метода на основании муравьиного алгоритма для выбранного класса данных;
- провести сравнительный анализ двух методов;
- дать рекомендации о применимости метода решения задачи коммивояжера на основе муравьиного алгоритма.

1 Аналитический раздел

Задача коммивояжёра — 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.

Задача коммивояжёра относится к числу транс вычислительных: 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.

Муравьиный алгоритм — один из эффективных полиномиальных алгоритмов 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.

1.1 Описание задачи

Задача коммивояжёра заключается 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.

Дано:

- M — число городов;
- D — матрица смежности.

Найти:

- Минимальный маршрут, проходящий через все города только 1 раз.

1.2 Описание алгоритма "Рекурсивный полный перебор"

Полный перебор — метод решения математических задач. Относится к классу методов поиска решения исчерпыванием всевозможных вариантов. Сложность полного перебора зависит от количества всех возможных решений задачи. Если пространство решений очень велико, то полный перебор может не дать результатов в течение нескольких лет или даже столетий. Полный перебор гарантировано дает идеальное решение, так как гарантируется, что каждый вариант будет рассмотрен.

Однако на практике этот алгоритм не применяется, так как чаще необходимо получить, возможно, не самое лучшее решение, но за минимальное время.

1.3 Описание алгоритма "Муравьиный алгоритм"

Идея муравьиной оптимизации — моделирование поведения муравьев, связанного с их способностью быстро находить кратчайший путь. При своем движении муравей помечает путь феромоном, и эта информация используется другими муравьями для выбора пути. Это элементарное правило поведения и определяет способность муравьев находить новый путь, если старый оказывается недоступным.

По сравнению с точными методами, например динамическим программированием или методом ветвей и границ, муравьиный алгоритм находит близкие к оптимуму решения за значительно меньшее время даже для задач небольшой размерности ($n > 20$). Время оптимизации муравьиным алгоритмом является полиномиальной функцией от размерности $O(t, n^2, m)$ [?], тогда как для точных методов зависимость экспоненциальная.

Каждый муравей выходит из своего города i , пройдя по своему маршруту, подсчитывает стоимость пройденного маршрута. Вероятность перехода из вершины i в вершину j определяется по формуле (1.1).

$$P_{ij,k}(t) = \begin{cases} \frac{[\tau_{ij}(t)]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{l \in J_{i,k}} [\tau_{il}(t)]^\alpha \cdot \eta_{il}^\beta} & , j \in J_{i,k}; \\ 0 & , j \notin J_{i,k} \end{cases} \quad (1.1)$$

где

$\eta_{i,j}$ — величина, обратная расстоянию от города i до j ;

$\tau_{i,j}$ — количество феромонов на ребре ij ;

β — параметр влияния длины пути;

α — параметр влияния феромона.

После завершения маршрута каждый муравей k откладывает на ребре (i, j) такое количество феромона (формула (1.2)):

$$\Delta\tau_{i,j}^k = \begin{cases} \frac{Q}{L_k} & \text{Если } k\text{-ый муравей прошел по ребру } ij; \\ 0 & \text{Иначе} \end{cases} \quad (1.2)$$

где

Q — количество феромона, переносимого муравьем;

L_k — стоимость маршрута k -го муравья

Для исследования всего пространства решений необходимо обеспечить испарение феромона — уменьшение во времени количества отложенного на предыдущих итерациях феромона. После окончания условного дня наступает условная ночь, в течение которой феромон испаряется с ребер с коэффициентом ρ . Правило обновления феромона примет вид в соответствии с формулой (1.3):

$$\tau_{i,j}(t+1) = (1 - \rho)\tau_{i,j}(t) + \Delta\tau_{i,j}(t), \quad (1.3)$$

где

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

$\tau_{i,j}(t)$ — количество феромона на дуге ij ;

$\Delta\tau_{i,j}(t)$ — количество отложенного феромона

t — номер текущего дня

$$\tau_{i,j}(t) = \sum_{k=1}^m \Delta\tau_{i,j}^k(t)$$

m — количество муравьев

1.4 Вывод

В данном разделе была описана суть задачи коммивояжера и описаны алгоритмы, которые будут использоваться в данной работе.

2 Конструкторский раздел

В данном разделе представлены схемы алгоритмов рекурсивного полного перебора и муравьиного алгоритма.

2.1 Схемы алгоритмов

Схема алгоритма рекурсивного полного перебора представлена на рисунке 2.1. Схема муравьиного алгоритма представлена на рисунке 2.2.

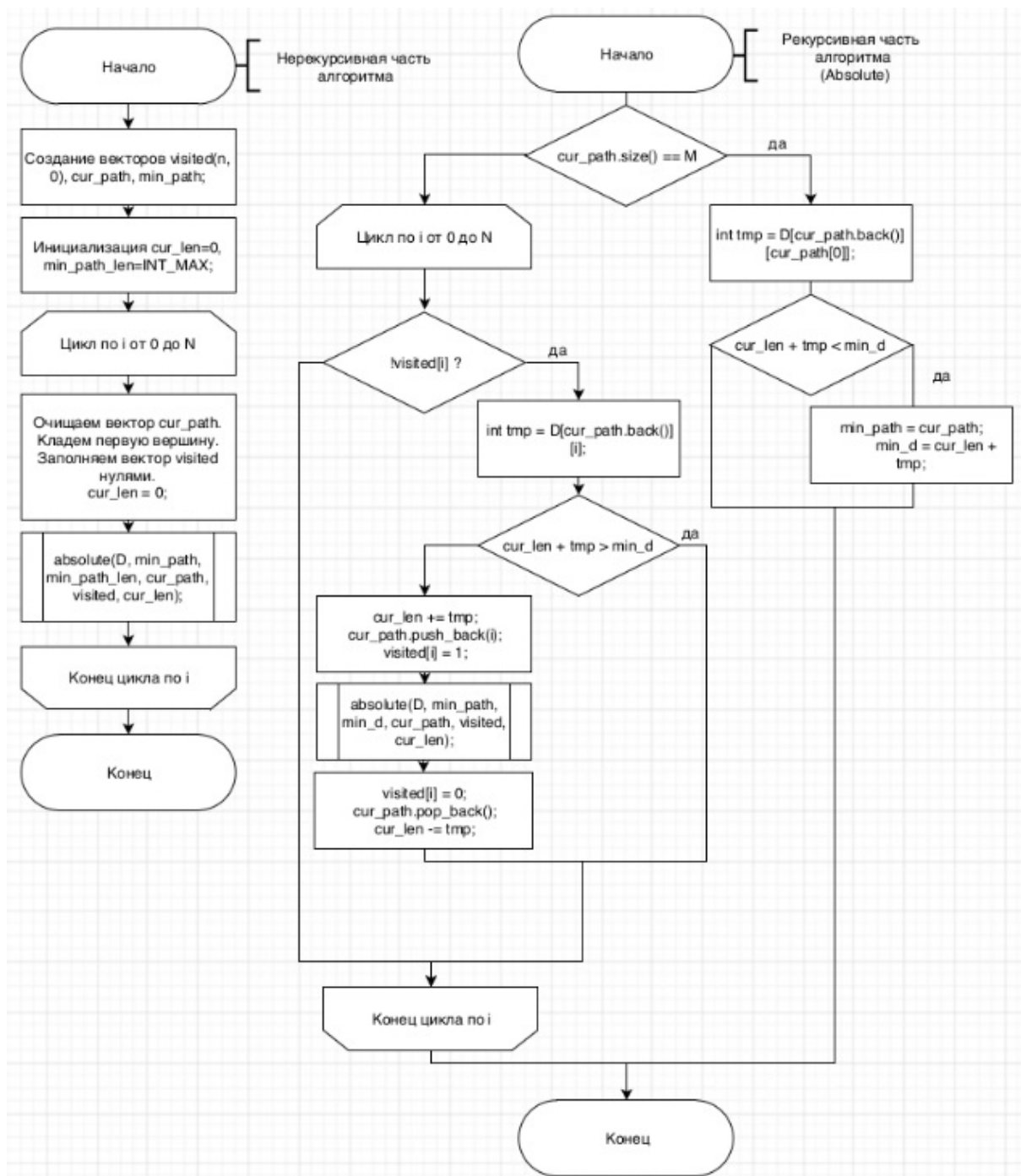


Рис. 2.1: Схема алгоритма рекурсивного полного перебора

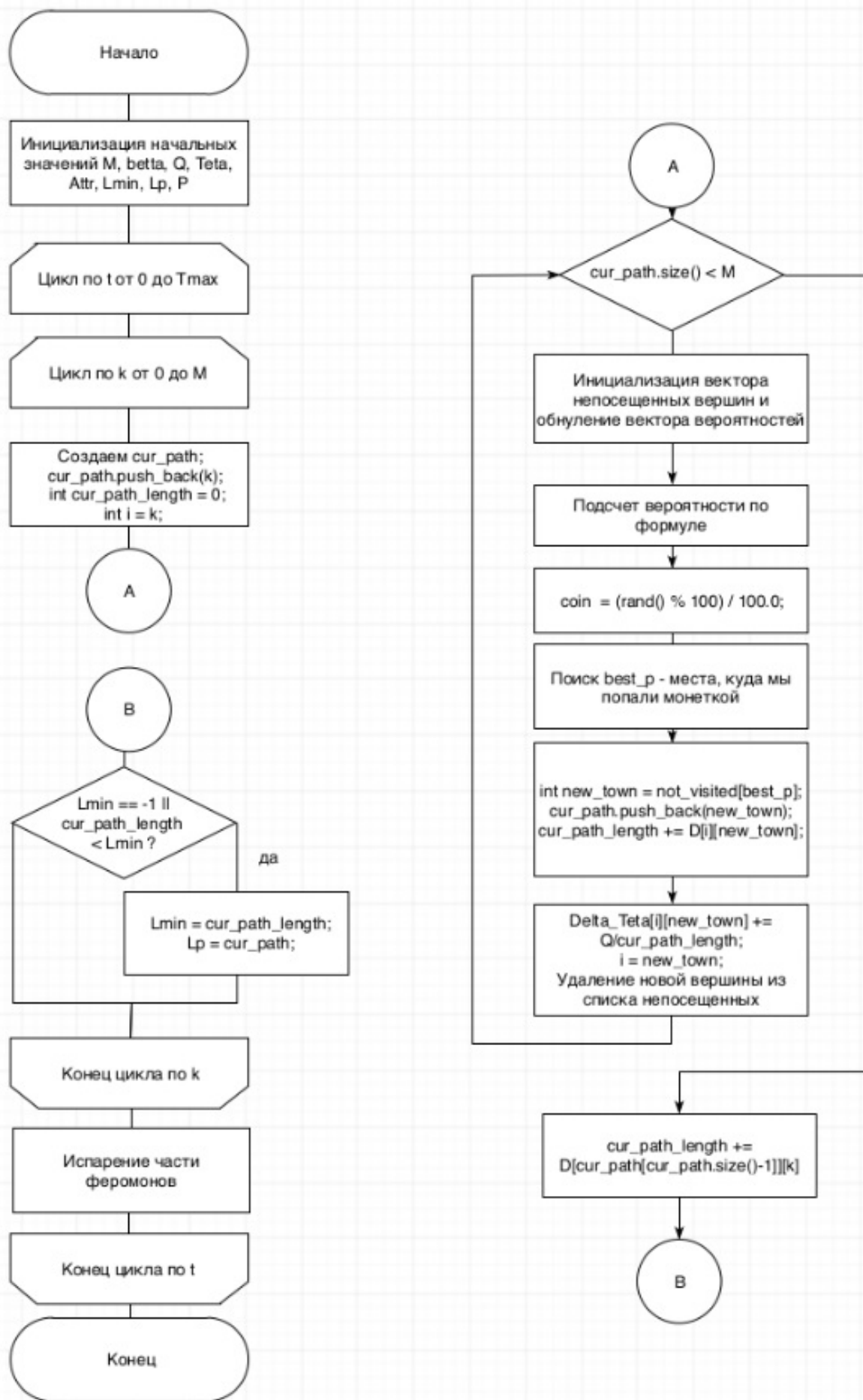


Рис. 2.2: Схема муравьиного алгоритма

2.2 Вывод

В данном разделе были разработаны схемы алгоритмов рекурсивного полного перебора и муравьиного алгоритма.

3 Технологический раздел

В данном разделе представлены требования к ПО, выбор инструментов для реализации и оценки алгоритмов, а также листинги полученного кода.

3.1 Требование к ПО

К программе предъявляется ряд требований:

- на вход программа получает количество вершин и матрицу смежностей;
- на выходе должен быть получен минимальный путь, проходящий через все вершины графа.

3.2 Выбор инструментов

Поскольку наиболее освоенным языком для разработчика является `C++`, для реализации поставленной задачи был выбран именно он, т.к. таким образом работа будет проделана наиболее быстро и качественно.

Соответственно для компиляции кода будет использоваться компилятор `G++`.

Чтобы оценить время выполнения программы будет замеряться реальное время, т.к. таким образом можно будет сравнить реализации алгоритмов без и с использованием параллельных вычислений. Для замера реального времени работы программы используется функция `clock()` т.к. программа тестируется на компьютере с установленной ОС Windows. [1]

Кроме этого, необходимо отключить оптимизации компилятора для более честного сравнения алгоритмов. В моём случае это делается с помощью ключа `-O0` т.к. используется компилятор `G++`. [2]

3.3 Реализация алгоритмов

На листингах 3.1-3.4 представлены реализации алгоритма рекурсивного полного перебора и муравьиного алгоритма.

Листинг 3.1: Реализация полного перебора, часть 1

```
1 pair<int, vector<int> > absolute_find(Matrix D)
2 {
3     int n = D.rows();
4     vector<bool> visited(n, 0);
5     vector<int> cur_path;
6     vector<int> min_path;
7     int cur_len = 0;
8     int min_path_len = INT_MAX;
9     for (int i = 0; i < n; i++)
10    {
11        cur_path.clear();
12        cur_path.push_back(i);
13        fill(visited.begin(), visited.end(), 0);
14        visited[i] = 1;
15        cur_len = 0;
16        absolute(D, min_path, min_path_len, cur_path, visited, cur_len);
17    }
18    return pair<int, vector<int>>(min_path_len, min_path);
19 }
```

Листинг 3.2: Реализация полного перебора, часть 2

```
1 void absolute(Matrix &D, vector<int> &min_path, int &min_d, vector<int> &cur_path,  
  vector<bool> &visited, int &cur_len) {  
2     size_t M = D.cols();  
3     if (cur_path.size() == M) {  
4         ss++;  
5         int tmp = D[cur_path.back()][cur_path[0]];  
6         if (cur_len + tmp < min_d)  
7         {  
8             min_path = cur_path;  
9             min_d = cur_len + tmp;  
10        }  
11        return;  
12    }  
13    for (size_t i = 0; i < M; i++) {  
14        if (!visited[i])  
15        {  
16            int tmp = D[cur_path.back()][i];  
17            if (cur_len + tmp > min_d)  
18                continue;  
19            cur_len += tmp;  
20            cur_path.push_back(i);  
21            visited[i] = 1;  
22            absolute(D, min_path, min_d, cur_path, visited, cur_len);  
23            visited[i] = 0;  
24            cur_path.pop_back();  
25            cur_len -= tmp;  
26        }  
27    }  
28 }
```

Листинг 3.3: Реализация муравьиного алгоритма, часть 1

```

1 pair<int, vector<int>> Ant(Matrix D, const int Tmax, const double alpha, const double
    ro){
2     const int teta_start = 10;
3     const int teta_min = 5;
4     const size_t M = D.cols(); //количество вершин
5     const double Q = D.avg()*M; //примерное значение длины пути
6     const double betta = A_B_CONST - alpha;
7
8     Matrix Attr(M, M); //привлекательность
9     get_attractiveness(Attr, D);
10    Matrix Teta(M, M, teta_start); //феромоны
11    Matrix Delta_Teta(M, M); //феромоны на тек шаге
12    int Lmin = -1; //длина кратчайшего маршрута
13    vector<int> Lp; // кратчайший маршрут
14    vector<double> P(M, 0.0); //вероятности
15
16    double coin;
17    // цикл по времени жизни колонии
18    for (int t = 0; t < Tmax; t++) {
19        Delta_Teta.zero();
20        //цикл по всем муравьям
21        for (size_t k = 0; k < M; k++) {
22            vector<int> cur_path;
23            cur_path.push_back(k);
24            int cur_path_length = 0;
25            int i = k;
26            //строим маршрут
27            while (cur_path.size() < M) {
28                vector<int> not_visited;
29                find_not_visited(not_visited, cur_path, M);
30                for (size_t j = 0; j < M ; j++){
31                    P[j] = 0.0;
32                }
33                // посчитаем вероятности
34                for (size_t j = 0; j < not_visited.size(); j++) {
35                    if (D[i][not_visited[j]]) {
36                        double sum = 0;
37                        for (auto n: not_visited) {
38                            sum += pow(Teta[i][n], alpha) * pow(Attr[i][n], betta);
39                        }
40                        P[j] = pow(Teta[i][not_visited[j]],
                                    alpha)*pow(Attr[i][not_visited[j]], betta)/sum;
41                    }
42                    else {
43                        P[j] = 0;
44                    }
45                }

```

Листинг 3.4: Реализация муравьиного алгоритма, часть 2

```

1      //подбросим монетку
2      coin = (rand() % 100) / 100.0;
3      // подсчитаем куда мы попали
4      int best_p = 0;
5      double sum_p = 0;
6      for (size_t s = 0; s < M; s++) {
7          sum_p += P[s];
8          if(coin < sum_p)
9              {
10                 best_p = s;
11                 break;
12             }
13     }
14     // добавим новый город в путь
15     int new_town = not_visited[best_p];
16     cur_path.push_back(new_town);
17     cur_path_length += D[i][new_town];
18     //обновим феромон на этом участке
19     Delta_Teta[i][new_town] += Q/cur_path_length;
20     i = new_town; // дальше продолжим путь из этого города
21     not_visited.erase(not_visited.begin()+best_p);
22 }
23 // конец построения маршрута
24 // осталось только добавить путь от последнего города к начальному
25 // это завершит цикл
26 cur_path_length += D[cur_path[cur_path.size()-1]][k];
27 // проверим не каратчайший ли это путь
28 if (Lmin == -1 || cur_path_length < Lmin){
29     Lmin = cur_path_length;
30     Lp = cur_path;
31 }
32 }
33 // конец цикла по муравьям
34 //теперь чатсь феромона должна испариться
35 for (size_t ii = 0; ii < M; ii++) {
36     for (size_t jj = 0; jj < M; jj++) {
37         Teta[ii][jj] = Teta[ii][jj] * (1.0 - ro) + Delta_Teta[ii][jj];
38         if (Teta[ii][jj] < teta_min)
39             {
40                 Teta[ii][jj] = teta_min;
41             }
42     }
43 }
44 }
45 // конец цикла по времени
46 return pair<int, vector<int>>(Lmin, Lp);
47 }

```


3.4 Тестирование

Тестирование проводилось вручную по методу чёрного ящика. Результаты представлены на рисунках 3.1-3.2.

```
Matrix:
0 2 1 2
2 0 2 3
1 2 0 2
2 3 2 0
Absolute result:
D = 8;
path: 0 1 2 3 ;
Ant result:
D = 8;
path: 0 2 3 1 ;
```

Рис. 3.1: Результаты тестирования 1

```
Matrix:
0 1 1 1 1
1 0 1 1 1
1 1 0 2 1
1 1 2 0 1
1 1 1 1 0
Absolute result:
D = 5;
path: 0 1 2 4 3 ;
Ant result:
D = 5;
path: 0 2 4 1 3 ;
```

Рис. 3.2: Результаты тестирования 2

Как видно, в первом тесте получены одинаковые результаты, соответствующие ожидаемым. Цикл начинается с разных вершин для обоих алгоритмов, однако так как это цикл, циклическим сдвигом влево вершин легко получается такой же цикл.

Во втором тесте пути отличаются сильно. Так как изначально существует несколько путей, оба алгоритма отработали верно.

3.5 Вывод

В данном разделе были выдвинуты требования к ПО, выбраны инструменты реализации выбранных алгоритмов, представлены листинги реализованных алгоритмов, а также проведено тестирование.

4 Исследовательский раздел

В данном разделе представлены технические характеристики компьютера, используемого для тестирования и экспериментов, и примеры работы программы.

4.1 Технические характеристики

Ниже приведены технические характеристики устройства, на котором были проведены эксперименты при помощи разработанного ПО:

- операционная система: Windows 10 (64-разрядная);
- оперативная память: 32 GB;
- процессор: Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz;
- количество ядер: 4;
- количество потоков: 8.

4.2 Примеры работы программы

На рисунках 4.1-4.2 представлены примеры работы программы.

```
Matrix:
0 6 9 8 5 9 2
6 0 4 1 8 3 9
9 4 0 3 8 7 8
8 1 3 0 6 8 9
5 8 8 6 0 4 1
9 3 7 8 4 0 1
2 9 8 9 1 1 0
Absolute result:
D = 23;
path: 0 2 3 1 5 4 6 ;
Ant result:
D = 23;
path: 3 1 5 4 6 0 2 ;
```

Рис. 4.1: Результат работы программы

```

Matrix:
0 6 9 8 5 9 2 4 1 8 3 9 3 8
6 0 7 8 6 8 9 4 1 1 7 6 1 5
9 7 0 8 7 6 9 6 3 1 3 1 7 5
8 8 8 0 9 2 8 4 3 7 3 4 7 3
5 6 7 9 0 4 8 3 2 6 6 2 7 4
9 8 6 2 4 0 8 3 4 8 5 5 3 6
2 9 9 8 8 8 0 7 1 2 5 6 5 5
4 4 6 4 3 3 7 0 6 1 6 7 8 6
1 1 3 3 2 4 1 6 0 4 7 4 3 1
8 1 1 7 6 8 2 1 4 0 6 1 2 1
3 7 3 3 6 5 5 6 7 6 0 6 8 6
9 6 1 4 2 5 6 7 4 1 6 0 9 2
3 1 7 7 7 3 5 8 3 2 8 9 0 7
8 5 5 3 4 6 5 6 1 1 6 2 7 0
Absolute result:
D = 27;
path: 0 6 8 1 12 5 3 13 9 7 4 11 2 10 ;
Ant result:
D = 28;
path: 6 0 12 1 9 7 5 3 10 2 11 4 13 8 ;

```

Рис. 4.2: Результат работы программы

4.3 Постановка экспериментов

Проведем эксперимент, чтобы установить при каких параметрах муравьиный алгоритм работает лучше всего. Возьмем значения γ_0 от 0 до 1 с шагом 0.25, значения α от 0 до 1 с шагом 0.25 и значения T_{\max} от 50 до 400 с шагом 50.

На рисунке 4.3 представлены результаты эксперимента. В первых трех столбцах находятся значения γ_0 , α и t_{\max} соответственно. В четвертом столбце находится среднее максимальное отклонение при данных параметрах. Замеры проводились для матрицы 12x12, также предварительно был высчитан идеальный путь с помощью алгоритма полного перебора.

1	ro	Alpha	tmax	Difference
2	0.250000	0.750000	400	0.000000
3	0.000000	0.750000	350	0.020000
4	0.000000	0.750000	400	0.020000
5	0.250000	0.750000	350	0.040000
6	0.250000	0.750000	300	0.060000
7	0.000000	0.750000	300	0.100000
8	0.250000	0.750000	250	0.100000
9	0.250000	0.750000	200	0.100000
10	0.000000	0.500000	400	0.140000
11	0.250000	1.000000	400	0.180000
12	0.000000	1.000000	350	0.180000
13	0.250000	1.000000	350	0.180000
14	0.250000	0.500000	400	0.200000
15	0.000000	0.750000	250	0.220000
16	0.250000	1.000000	300	0.220000

Рис. 4.3: Результат эксперимента с изменением параметров

По полученным данным можно сделать вывод, что алгоритм лучше всего работает на значениях ro от 0 до 0.25 и alpha от 0.75 до 1 при tmax от 350 до 400.

Далее проведем замеры по времени для муравьиного алгоритма с параметрами ro=0.25, alpha=0.75, tmax = 350 и для алгоритма полного перебора.

Для произведения замеров времени выполнения реализаций алгоритмов будет использована формула $t = \frac{T_n}{N}$, где N – количество замеров, t – время выполнения реализации алгоритма, T_n – время выполнения N замеров. Неоднократное измерение времени необходимо для построения более гладкого графика и получения усредненного значения времени.

Количество замеров будет взято равным 50. Замеры проводятся для матриц размером от 2 до 15 элементов.

На рисунке 4.4 представлены результаты эксперимента.



Рис. 4.4: Зависимость времени работы алгоритмов от размера матрицы

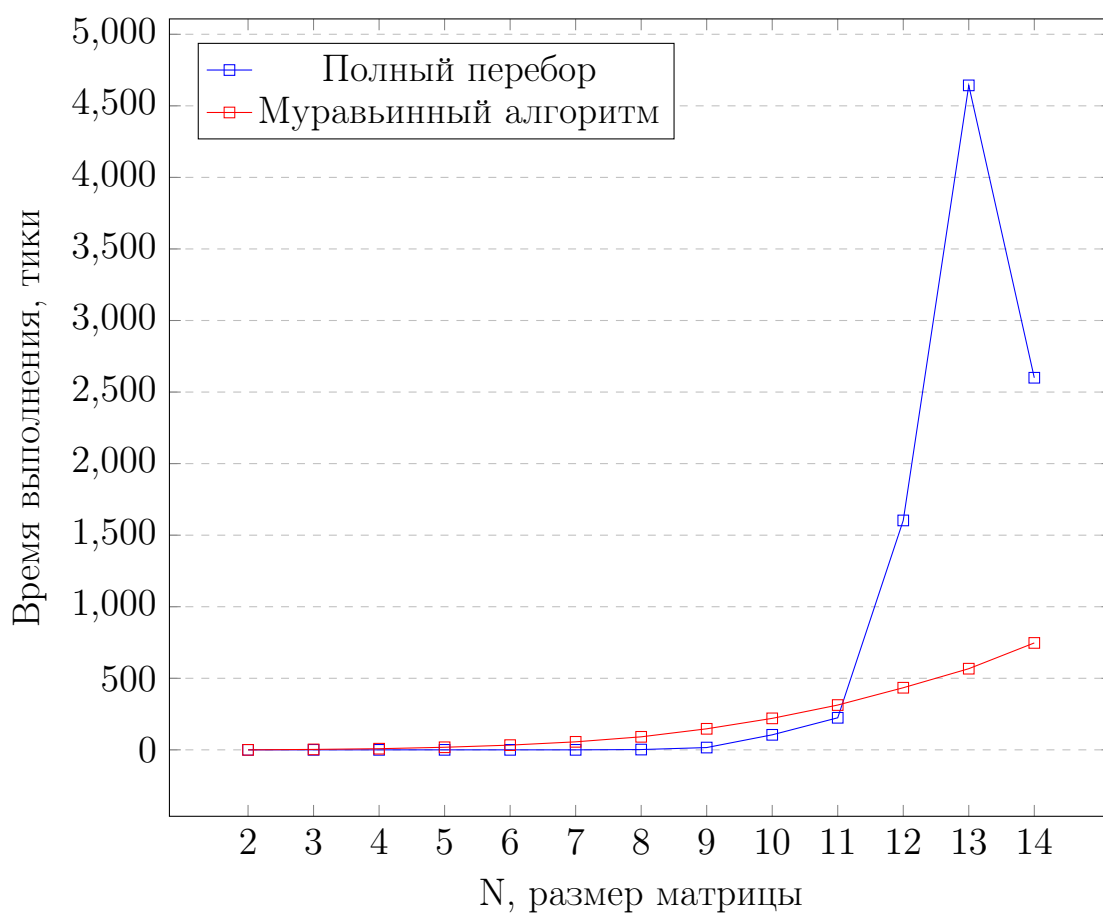


Рис. 4.5: Зависимость времени работы алгоритмов от размера матрицы

По рисунку видно, что алгоритм полного перебора работает гораздо медленнее, чем муравьиный алгоритм, начиная с матрицы размером 10x10 и далее.

4.4 Вывод

По итогу исследования выяснилось, что разработанный алгоритм работает верно, то есть находит самый длинный полиндром в строке. Кроме этого был проведён анализ и сделан вывод по логу программы.

В ходе экспериментов было вычислены параметры муравьиного алгоритма, при которых результат наиболее приближен к идеальному. При увеличении параметра t_{max} возрастает вероятность найти оптимальное решение, однако возрастает и время работы программы. По скорости муравьиный алгоритм сильно выигрывает у алгоритма полного перебора, на матрицах, размер которых превышает 10x10. Алгоритм полного перебора следует использовать на матрицах меньшего размера или в задачах, где требуется гарантировано точное решение.

Заключение

По итогу проделанной работы была достигнута цель – изучен муравьиный алгоритм на материале решения задачи коммивояжёра.

Также были решены все поставленные задачи, а именно:

- описаны методы решения;
- описана реализация, реализован метод;
- выбран класс данных, составлен набор данных;
- проведена параметризация метода на основании муравьиного алгоритма для выбранного класса данных;
- проведён сравнительный анализ двух методов;
- даны рекомендации о применимости метода решения задачи коммивояжера на основе муравьиного алгоритма.

Список использованных источников

- [1] MSDN – библиотека <clock> [Электронный ресурс] // Техническая документация Майкрософт. URL: <https://docs.microsoft.com/ru-ru/cpp/c-runtime-library/reference/clock?view=msvc-160&viewFallbackFrom=vs-2019> (дата обращения: 13.12.21)
- [2] Как применить настройки оптимизации GCC в QT? [Электронный ресурс] // Блог ПерСа. URL: <http://blog.kislenko.net/show.php?id=1991> (дата обращения: 07.12.21)