



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 ИУ, Информатика и системы управления

КАФЕДРА ИУ7, Программное обеспечение ЭВМ и информационные технологии

# ЛАБОРАТОРНАЯ РАБОТА №6

## *ПО ДИСЦИПЛИНЕ*

### *“Анализ алгоритмов”*

Студент      ИУ7-54Б  
(Группа)

\_\_\_\_\_  
(Подпись, дата)      **А.А. Андреев**  
(И.О.Фамилия)

Преподаватель

\_\_\_\_\_  
(Подпись, дата)      **Л.Л. Волкова**  
(И.О.Фамилия)

2021 г.

# **Содержание**

<b>Введение</b>	<b>3</b>
<b>1. Аналитическая часть</b>	<b>4</b>
1.1 Задача коммивояжера	4
1.2 Описание муравьиного алгоритма	4
Вывод	7
<b>2. Конструкторская часть.</b>	<b>8</b>
2.1 Разработка алгоритмов	8
Вывод	8
<b>3. Технологическая часть.</b>	<b>9</b>
3.1 Требования к программному обеспечению	9
3.2 Средства реализации	9
3.3 Реализация алгоритмов	9
Вывод	16
<b>4. Исследовательская часть.</b>	<b>17</b>
4.1 Демонстрация работы программы	17
4.2 Технические характеристики	17
4.3 Время выполнения алгоритмов	18
Вывод	20
<b>Заключение</b>	<b>21</b>
<b>Список литературы</b>	<b>22</b>

## **Введение**

Данная лабораторная работа посвящена исследованию муравьиного алгоритма.

Имитация самоорганизации муравьиной колонии составляет основу муравьиных алгоритмов оптимизации — нового перспективного метода природных вычислений. Колония муравьев может рассматриваться как много-агентная система, в которой каждый агент (муравей) функционирует автономно по очень простым правилам. В противовес почти примитивному поведению агентов, поведение всей системы получается на удивление разумным.

Муравьиные алгоритмы серьезно исследуются европейскими учеными с середины 90х годов. На сегодня уже получены хорошие результаты муравьиной оптимизации таких сложных комбинаторных задач, как: задачи коммивояжера, задачи оптимизации маршрутов грузовиков, задачи раскраски графа, квадратичной задачи о назначениях, оптимизации сетевых графиков, задачи календарного планирования и других. Особенно эффективны муравьиные алгоритмы при online-оптимизации процессов в распределенных нестационарных системах, например трафиков в телекоммуникационных сетях [1].

**Цель данной работы является исследование особенностей работы муравьиного алгоритма.**

**Для достижения поставленной цели необходимо выполнить следующие задачи:**

1. Реализовать муравьиный алгоритм на основе теоретических знаний;
2. Получить практические навыки во время реализации;
3. Экспериментально подтвердить различия во временной эффективности работы муравьиного алгоритма при разных значениях коэффициента важности величины пути и коэффициента важности мощности феромона при помощи разработанного программного обеспечения на материале замеров процессорного времени;
4. Описать и обосновать полученные результаты в отчете о выполненной лабораторной работе, выполненном как расчётно-пояснительная записка к работе.

# **1. Аналитическая часть**

В данном разделе будет описана теоретическая основа муравьиного алгоритма.

## **1.1 Задача коммивояжера**

Задача коммивояжера формулируется как задача поиска минимального по стоимости замкнутого маршрута по всем вершинам без повторений на полном взвешенном графе с  $n$  вершинами. Вершины графа являются городами, которые должен посетить коммивояжер, а веса ребер отражают расстояния или стоимости проезда. Эта задача является NP-трудной, и точный переборный алгоритм ее решения имеет факториальную сложность [2].

## **1.2 Описание муравьиного алгоритма**

Муравьиные алгоритмы представляют собой вероятностную жадную эвристику, где вероятности устанавливаются, исходя из информации о качестве решения, полученной из предыдущих решений. Идея муравьиного алгоритма - моделирование поведения муравьев, связанного с их способностью быстро находить кратчайший путь от муравейника к источнику пищи и адаптироваться к изменяющимся условиям, находя новый кратчайший путь.

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.

С учётом особенностей задачи коммивояжёра, мы можем описать локальные правила поведения муравьёв при выборе пути.

1. Муравьи обладают «памятью». Поскольку каждый город может быть посещён только один раз, то у каждого муравья есть список уже посещённых городов. Обозначим через  $J_{i,k}$  список городов, которые необходимо посетить муравью  $k$ , находящемуся в городе  $i$ .
2. Муравьи обладают «зрением», которое определяет степень желания посетить город  $j$ , если муравей находится в городе  $i$ . Будем считать, что видимость обратно пропорциональна расстоянию между городами.
3. Муравьи обладают «обонянием», с помощью которого они могут улавливать след феромона, подтверждающий желание посетить город  $j$  из города  $i$  на основании опыта других муравьёв. Количество феромона на ребре  $(i,j)$  в момент времени  $t$  обозначим через  $\tau_{ij}(t)$ .
4. На основании предыдущих утверждений мы можем сформулировать вероятностно-пропорциональное правило, определяющее вероятность перехода  $k$ -ого муравья из города  $i$  в город  $j$ :

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

где  $\tau_{ij}(t)$  – уровень феромона,  $\eta_{ij}(t)$  – эвристическое расстояние, а  $\alpha$  и  $\beta$  – константные параметры.

Выбор города является вероятностным, в общую зону всех городов бросается случайное число, которое и определяет выбор муравья. При  $\alpha = 0$  алгоритм вырождается до жадного алгоритма, по которому на каждом шаге будет выбираться ближайший город.

5. При прохождении ребра муравей оставляет на нём некоторое количество феромона, которое должно быть связано с оптимальностью сделанного выбора. Пусть есть маршрут, пройденный муравьём  $k$  к моменту времени  $t$ ,  $L$  – длина этого маршрута,  $L_k(t)$  - цена текущего решения для  $k$ -ого муравья а  $Q$  – параметр, имеющий значение порядка цены оптимального решения. Тогда откладываемое количество феромона

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

а испаряемое количество феромона

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

где  $m$  – количество муравьёв в колонии [3].

### 1.3 Вариации муравьиного алгоритма

Выделяют четыре 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 ( $t_{max}$ ,  $t_{min}$ ). Феромоны откладываются только на глобально лучших или лучших в итерации путях. Все рёбра инициализируются значением  $t_{max}$ .
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 (СОАС).** Механизм отложения феромонов СОАС позволяет муравьям искать решения 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.

#### Вывод

В аналитическом разделе была описана теоретическая основа муравьиного алгоритма, изучены различные его вариации со своими преимуществами и недостатками. Также описана задача коммивояжера.

## 2. Конструкторская часть.

В данном разделе будет приведены блок-схемы алгоритмов, описанных в аналитическом разделе.

### 2.1 Разработка алгоритмов

На рисунке 2.1 приведена обобщенная схема муравьиного алгоритма.

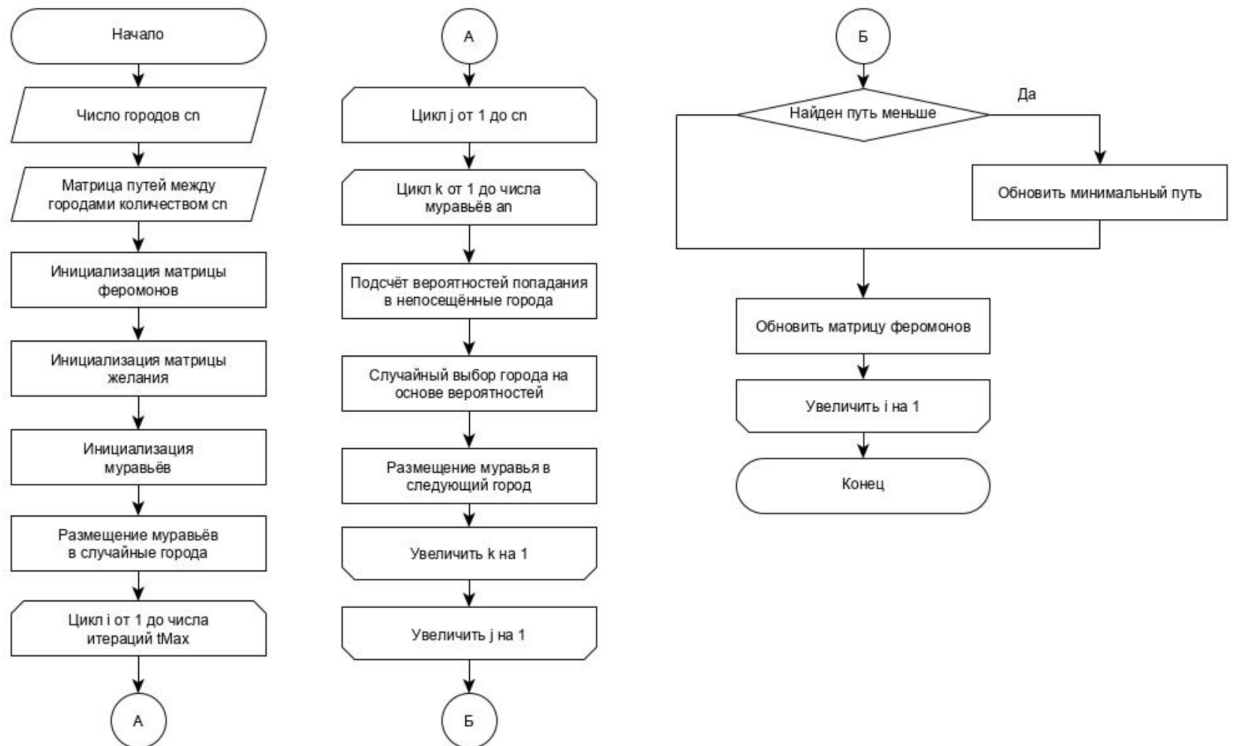


Рисунок 2.1: Схема муравьиного алгоритма

### Вывод

В данном разделе были рассмотрены блок-схемы, которые позволяют перейти к технологической части.



### 3. Технологическая часть.

В данном разделе будут рассмотрены требования к разрабатываемому программному обеспечению, средства, использованные в процессе разработки для реализации поставленных задач.

#### 3.1 Требования к программному обеспечению

Программное обеспечение должно реализовывать поставленную на лабораторную работу задачу. Интерфейс для взаимодействия с программой - командная строка. Пользователь должен иметь возможность вводить количество объектов, которые будут обрабатываться.

#### 3.2 Средства реализации

Для разработки данной программы применён язык C++ [2] с функцией `rdtsc()` из библиотеки `stdrin.h` [3] для вычисления времени работы процессора, чтобы расширить знания в области данного языка программирования.

#### 3.3 Реализация алгоритмов

В листингах 3.1-3.8 приведена реализация муравьиного алгоритма.

Листинг 3.1: Класс Муравья (Aco.hpp)

```
1. class Ant
2. {
3. public:
4.     explicit Ant(size_t graph_size);
5.
6.     size_t path_len;
7.     std::vector<bool> visited;
8.     std::vector<size_t> path;
9.
10.    void visit_city(const size_t city, const size_t
        cur_path_len,
11.                  const size_t cur_path_dist);
12.    void clear_visits();
13.    void make_default_path();
14.    bool is_visited(const size_t city) const;
15. };
16.
```

### Листинг 3.2: Методы класса Ant (Aco.cpp)

```
1. /// Ant
2.
3. Ant::Ant(size_t graph_size) : path_len(0)
4. {
5.     for (size_t i = 0; i < graph_size; i++)
6.     {
7.         path.push_back(0);
8.         visited.push_back(false);
9.     }
10. }
11.
12. void Ant::visit_city(const size_t city, const size_t
    cur_path_len,
13.     const size_t cur_path_dist)
14. {
15.     path_len += cur_path_dist;
16.     path[cur_path_len] = city;
17.     visited[city] = true;
18. }
19.
20. void Ant::clear_visits()
21. {
22.     for (size_t i = 0; i < visited.size(); i++)
23.         visited[i] = false;
24.     path_len = 0;
25. }
26.
27. void Ant::make_default_path()
28. {
29.     path_len = 0;
30.     visit_city(path[path.size() - 1], 0, 0);
31. }
32.
33. bool Ant::is_visited(const size_t city) const
34. {
35.     return visited[city];
36. }
```

### Листинг 3.3: Класс алгоритма ACO (Aco.hpp)

```
1. class ACO
2. {
3. private:
4.     const std::vector<std::vector<int>> dist_graph;
5.     const size_t cities_count;
6.
7.     std::vector<std::vector<double>> pher_graph;
8.     std::vector<std::vector<double>> desire_graph;
9.
10.    std::vector<Ant> ants;
11.    size_t ants_count;
12.
13.    std::vector<double> paths_probs;
14.
15.    double alpha = 0.5;
16.    double rho = 0.5;
17.    size_t tMax = 100;
18.    double beta = 1 - alpha;
19.
20.    const double Q = 5;
21.    const double ants_factor = 1;
22.    const double init_pher_value = 1;
23.
24. public:
25.     size_t min_len = 0;
26.     std::vector<size_t> min_path;
27.
28.     explicit ACO(const Graph<int>& graph);
29.
30.     void execute();
31.     void change_params(double alpha, double rho, size_t tMax);
32.
33. private:
34.     void make_default_state();
35.     void init_ants();
36.     void init_pher_graph();
37.     void pave_ants_paths();
38.     size_t get_next_city(const Ant& ant, const size_t
cur_city);
39.     void update_min_path();
40.     void update_pheromones();
41.     void make_default_ants();
42.     size_t select_next_city();
43.     double get_sum_probabilities();
44. };
```

### Листинг 3.4: Методы инициализации и запуска алгоритма (Aco.hpp), Часть 1

```

1. // ACO
2.
3. ACO::ACO(const Graph<int>& graph) :
4.     dist_graph(graph.graph), cities_count(graph.size)
5. {
6.     // init pher_graph
7.     for (size_t i = 0; i < cities_count; i++)
8.     {
9.         std::vector<double> line;
10.        for (size_t j = 0; j < cities_count; j++)
11.            line.push_back(init_pher_value);
12.        pher_graph.push_back(line);
13.    }
14.
15.    // init desire_graph
16.    for (size_t i = 0; i < cities_count; i++)
17.    {
18.        std::vector<double> line;
19.        for (size_t j = 0; j < cities_count; j++)
20.            line.push_back(dist_graph[i][j] == 0 ? 0 :
21.                            1.0 / dist_graph[i][j]);
22.        desire_graph.push_back(line);
23.    }
24.
25.    // init ants_count
26.    ants_count = cities_count * ants_factor;
27.    for (size_t i = 0; i < ants_count; i++)
28.    {
29.        Ant ant(cities_count);
30.        ants.push_back(ant);
31.    }
32.
33.    // init paths_probs
34.    for (size_t i = 0; i < cities_count; i++)
35.        paths_probs.push_back(0);
36. }
37.
38. void ACO::execute()
39. {
40.     make_default_state();
41.     init_pher_graph();
42.     init_ants();
43.
44.     for (size_t i = 0; i < tMax; i++)
45.     {
46.         pave_ants_paths();
47.         update_min_path();
48.         update_pheromones();
49.         make_default_ants();
50.     }
51. }
52.

```

### Листинг 3.5: Методы инициализации и запуска алгоритма (Aco.hpp), Часть 2

```
53. void ACO::change_params(double alpha, double rho, size_t
    tMax)
54. {
55.     this->alpha = alpha;
56.     this->beta = 1 - alpha;
57.     this->rho = rho;
58.     this->tMax = tMax;
59. }
60.
61. void ACO::make_default_state()
62. {
63.     min_len = 0;
64.     min_path.clear();
65. }
66.
67. void ACO::init_ants()
68. {
69.     for (size_t i = 0; i < ants_count; i++)
70.     {
71.         ants[i].clear_visits();
72.         ants[i].visit_city(rand() % cities_count, 0, 0);
73.     }
74. }
75.
76. void ACO::init_pher_graph()
77. {
78.     for (size_t i = 0; i < cities_count; i++)
79.         for (size_t j = 0; j < cities_count; j++)
80.             pher_graph[i][j] = init_pher_value;
81. }
```

### Листинг 3.6: Основные функции муравьиного алгоритма (Aco.hpp), Часть 1

```

1. void ACO::pave_ants_paths()
2. {
3.     for (size_t i = 0; i < cities_count - 1; i++)
4.     {
5.         for (size_t j = 0; j < ants_count; j++)
6.         {
7.             const size_t cur_city = ants[j].path[i];
8.             const size_t next_city = get_next_city(ants[j],
9.             cur_city);
10.            const int dist = dist_graph[cur_city][next_city];
11.            ants[j].visit_city(next_city, i + 1, dist);
12.        }
13.    }
14.
15.    for (size_t j = 0; j < ants_count; j++)
16.    {
17.        size_t i_ind = ants[j].path[ants[j].path.size() -
18.        1];
19.        size_t j_ind = ants[j].path[0];
20.        const int dist_init_city = dist_graph[i_ind][j_ind];
21.        ants[j].path_len += dist_init_city;
22.    }
23.
24.    size_t ACO::get_next_city(const Ant& ant, const size_t
25.    cur_city)
26.    {
27.        double sumP = 0;
28.        for (size_t i = 0; i < cities_count; i++)
29.        {
30.            double pher_factor = pow(pher_graph[cur_city][i],
31.            alpha);
32.            double desire_factor =
33.            pow(desire_graph[cur_city][i], beta);
34.            sumP += pher_factor * desire_factor;
35.        }
36.        for (size_t i = 0; i < cities_count; i++)
37.        {
38.            if (i == cur_city || ant.is_visited(i))
39.                paths_probs[i] = 0;
40.            else
41.            {
42.                double pher_factor =
43.                pow(pher_graph[cur_city][i], alpha);
44.                double desire_factor =
45.                pow(desire_graph[cur_city][i], beta);
46.                paths_probs[i] = pher_factor * desire_factor /
47.                sumP;
48.            }
49.        }
50.        return select_next_city();
51.    }
52. }

```

```
47. }
```

Листинг 3.7: Основные функции муравьиного алгоритма (Aco.hpp), Часть 2

```
48. void ACO::update_min_path()
49. {
50.     for (size_t i = 0; i < ants_count; i++)
51.     {
52.         const size_t cur_len = ants[i].path_len;
53.         if (cur_len < min_len || min_len == 0)
54.         {
55.             min_len = cur_len;
56.             min_path = ants[i].path;
57.         }
58.     }
59. }
60.
61. void ACO::update_pheromones()
62. {
63.     for (size_t i = 0; i < cities_count; i++)
64.         for (size_t j = 0; j < cities_count; j++)
65.             pher_graph[i][j] *= (1 - rho);
66.
67.     for (size_t i = 0; i < ants_count; i++)
68.     {
69.         Ant& ant = ants[i];
70.
71.         double dt = Q / ant.path_len;
72.         for (size_t j = 0; j < cities_count - 1; j++)
73.             pher_graph[ant.path[j]][ant.path[j + 1]] += dt;
74.         pher_graph[ant.path[cities_count - 1]][ant.path[0]]
+= dt;
75.     }
76. }
77.
78. void ACO::make_default_ants()
79. {
80.     for (size_t i = 0; i < ants_count; i++)
81.     {
82.         ants[i].clear_visits();
83.         ants[i].make_default_path();
84.     }
85. }
```

### Листинг 3.8: Основные функции муравьиного алгоритма (Aco.hpp), Часть 3

```
86.  size_t ACO::select_next_city()
87.  {
88.      double sum_probabilities = get_sum_probabilities();
89.      double rand_num = ((double) rand() / (RAND_MAX)) *
sum_probabilities;
90.      double total = 0;
91.      size_t city = 0;
92.
93.      for (size_t i = 0; i < cities_count && total < rand_num;
i++)
94.      {
95.          total += paths_probs[i];
96.          if (total >= rand_num)
97.              city = i;
98.      }
99.
100.     return city;
101. }
102.
103. double ACO::get_sum_probabilities()
104. {
105.     double sum_probabilities = 0;
106.     for (size_t i = 0; i < cities_count; i++)
107.         sum_probabilities += paths_probs[i];
108.     return sum_probabilities;
109. }
```

### Вывод

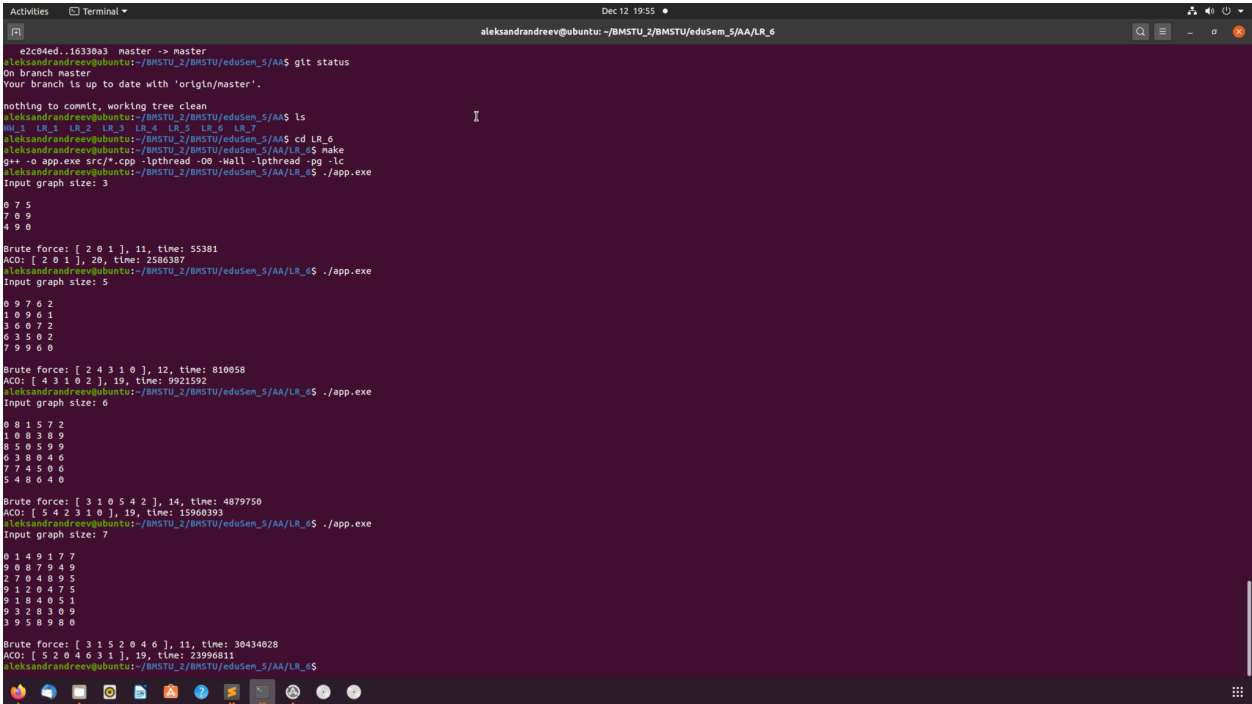
В данном разделе были рассмотрены требования к разрабатываемому программному обеспечению, средства, использованные в процессе разработки для реализации поставленных задач.



## 4. Исследовательская часть.

### 4.1 Демонстрация работы программы

Пример работы программы представлен на рисунке 4.1.



```
e2c04ed..16330a3 master -> master
aleksandrareev@ubuntu:~/BMSTU_2/BMSTU/eduSem_5/AA$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
aleksandrareev@ubuntu:~/BMSTU_2/BMSTU/eduSem_5/AA$ ls
LR_1 LR_2 LR_3 LR_4 LR_5 LR_6 LR_7
aleksandrareev@ubuntu:~/BMSTU_2/BMSTU/eduSem_5/AA$ cd LR_6
aleksandrareev@ubuntu:~/BMSTU_2/BMSTU/eduSem_5/AA/LR_6$ make
g++ -o app.exe src/*.cpp -lpthread -DGO -Wall -lpthread -pg -lc
aleksandrareev@ubuntu:~/BMSTU_2/BMSTU/eduSem_5/AA/LR_6$ ./app.exe
Input graph size: 3
0 7 5
7 0 9
4 9 0
Brute force: [ 2 0 1 ], 11, time: 55381
ACO: [ 2 0 1 ], 20, time: 2586387
aleksandrareev@ubuntu:~/BMSTU_2/BMSTU/eduSem_5/AA/LR_6$ ./app.exe
Input graph size: 5
0 9 7 6 2
1 0 9 6 1
3 6 0 7 2
6 3 5 0 2
7 9 9 6 0
Brute force: [ 2 4 3 1 0 ], 12, time: 810658
ACO: [ 4 3 1 0 2 ], 19, time: 9921592
aleksandrareev@ubuntu:~/BMSTU_2/BMSTU/eduSem_5/AA/LR_6$ ./app.exe
Input graph size: 6
0 8 1 5 7 2
1 0 8 3 8 9
8 5 0 5 9 9
6 3 8 0 4 6
7 7 4 5 0 6
5 4 8 6 4 0
Brute force: [ 3 1 0 5 4 2 ], 14, time: 4879750
ACO: [ 5 4 2 3 1 0 ], 19, time: 15960399
aleksandrareev@ubuntu:~/BMSTU_2/BMSTU/eduSem_5/AA/LR_6$ ./app.exe
Input graph size: 7
0 1 4 9 1 7 7
9 0 8 7 9 4 9
2 7 0 4 8 9 5
9 1 2 0 4 7 5
9 1 8 4 0 5 1
9 3 2 0 3 0 9
3 9 5 8 9 8 0
Brute force: [ 3 1 5 2 0 4 6 ], 11, time: 30434028
ACO: [ 5 2 0 4 6 3 1 ], 19, time: 23996811
aleksandrareev@ubuntu:~/BMSTU_2/BMSTU/eduSem_5/AA/LR_6$
```

Рисунок 4.1: Демонстрация работы программы

### 4.2 Технические характеристики

В Таблице 4.2 приведены технические характеристики ЭВМ, на котором проводилось тестирование разрабатываемого программного обеспечения.

Таблица 4.2: Технические характеристики ЭВМ, на котором проводилось тестирование разрабатываемого программного обеспечения

ОС	Mac OS Mojave 64-bit
ОЗУ	8 Gb 2133 MHz LPDDR3
Процессор	2,3 GHz Intel Core i5

### 4.3 Время выполнения алгоритмов

В Таблицах 4.3.1 и 4.3.2 приведена информация о времени выполнения алгоритмов на случайных данных в микросекундах. Каждый замер проводился 10 раз, результат усреднялся.

Таблица 4.3.1: Сравнение работы муравьиного алгоритма на различных параметрах (в микросекундах), Часть 1

№	$\alpha$	$\rho$	$T_{max}$	Минимальный путь
1	0	0	100	16
2	0	0	200	16
3	0	0,25	100	18
4	0	0,25	200	16
5	0	0,5	100	16
6	0	0,5	200	16
7	0	0,75	100	16
8	0	0,75	200	16
9	0	1	100	16
10	0	1	200	16
11	0,25	0	100	16
12	0,25	0	200	16
13	0,25	0,25	100	16
14	0,25	0,25	200	16
15	0,25	0,5	100	19
16	0,25	0,5	200	16
17	0,25	0,75	100	16
18	0,25	0,75	200	16
19	0,25	1	100	14
20	0,25	1	200	12
21	0,5	0	100	16
22	0,5	0	200	18
23	0,5	0,25	100	16

Таблица 4.3.1: Сравнение работы муравьиного алгоритма на различных параметрах (в микросекундах), Часть 2

№	$\alpha$	$\rho$	$T_{max}$	Минимальный путь
24	0,5	0,25	200	16
25	0,5	0,5	100	16
26	0,5	0,5	200	16
27	0,5	0,75	100	16
28	0,5	0,75	200	16
29	0,5	1	100	15
30	0,75	1	200	13
31	0,75	0	100	22
32	0,75	0	200	16
33	0,75	0,25	100	21
34	0,75	0,25	200	18
35	0,75	0,5	100	16
36	0,75	0,5	200	16
37	0,75	0,75	100	16
38	0,75	0,75	200	16
39	0,75	1	100	18
40	1	1	200	15
41	1	0	100	22
42	1	0	200	23
43	1	0,25	100	20
44	1	0,25	200	25
45	1	0,5	100	27
46	1	0,5	200	22
47	1	0,75	100	24
48	1	0,75	200	18

Таблица 4.3.1: Сравнение работы муравьиного алгоритма на различных параметрах (в микросекундах), Часть 3

№	$\alpha$	$\rho$	$T_{max}$	Минимальный путь
49	1	1	100	22
50	1	1	200	21

### Вывод

В данном разделе был проведен сравнительный анализ работы реализованного муравьиного алгоритма при различных параметрах, из которого можно сделать вывод, что при правильном подборе параметров муравьиный алгоритм находит оптимальный ответ за приемлимое время, намного отличающееся (на 99.6% быстрее на графе из 10 узлов) от времени нахождения пути полным перебором.

## **Заключение**

В ходе выполнения данной лабораторной работы были изучены принципы муравьиного алгоритма. Было проведено исследование работы алгоритма при различных параметрах, из которого можно сделать вывод, что при правильном подборе параметров муравьиный алгоритм находит оптимальный ответ за приемлимое время, намного отличающееся (на 99.6% быстрее на графе из 10 узлов) от времени нахождения пути полным перебором.

## Список литературы

- [1] Кормен Т.Х., Лейзерсон Ч.И., Алгоритмы: Построение и анализ, год выпуска 2019, тираж 1328, 700 страниц.
- [2] ISO/IEC JTC1 SC22 WG21 N 3690 «Programming Languages — C++» [Электронный ресурс]. <https://devdocs.io/cpp/> Дата обращения: 13.09.2021
- [3] Гасфилд, Строки, деревья и последовательности в алгоритмах. Информатика и вычислительная биология. Невский Диалект БВХ-Петербург, год выпуска 2003, тираж 900, 653 страницы.
- [4] Вычисление процессорного времени выполнения программы [Электронный ресурс] Режим доступа: [https://www.tutorialspoint.com/python/time\\_clock.htm](https://www.tutorialspoint.com/python/time_clock.htm). Дата обращения: 13.09.2021