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»

Отчёт

по лабораторной работе №6

Название «Муравьиный алгоритм и метод полного перебора для решения задачи коммивояжера»

Дисциплина «Анализ алгоритмов»

Студент ИУ7-55Б _____ Бугаенко А.П.
(подпись, дата) (Фамилия И.О.)

Преподователь _____ Волкова Л.Л.
(подпись, дата) (Фамилия И.О.)

Москва, 2021

Содержание

Введение

Задача коммивояжера — одна из самых известных задач комбинаторной оптимизации, заключающаяся в поиске самого выгодного маршрута, проходящего через указанный города хотя бы по одному разу с последующим возвратом в исходный город. [1] Стоит заметить, что использование методов вычисления, которые гарантированно находят самый лучший результат, например метод полного перебора всех путей, приводит к тому, что решение задачи таким способом невозможно в рамках текущих мощностей вычислительных машин. Поэтому для решения этой задачи используют эврестические алгоритмы, которые не могут дать абсолютно верного решения, однако позволяют получить результаты достаточно близкие к нему.

Муравьиный алгоритм — один из эффективных полиномиальных алгоритмов для нахождения приближенных решений задачи коммивояжера, а также решения аналогичных задач поиска маршрутов на графах. Суть подхода заключается в анализе и использовании модели поведения муравьев, ищущих пути от колонии к источнику питания, и представляет собой метаэвристическую оптимизацию. [2]

Целью данной работы является изучение муравьиного алгоритма на материале решения задачи коммивояжера. Для того, чтобы достичь поставленной цели, нам необходимо выполнить следующие задачи:

1) провести анализ алгоритмов полного перебора и муравьиного алгоритма;
2) описать используемые структуры данных;
3) привести схемы рассматриваемых алгоритмов;
4) программно реализовать описанные выше алгоритмы;
5) методом подбора найти оптимальные параметры для муравьиного алгоритма;
6) провести сравнительный анализ каждого алгоритма по затрачиваемому в процессе работы времени.

1 Аналитический раздел

В данном разделе будут рассмотрены теоретически основы работы алгоритмов полного перебора и муравьиного алгоритма для задачи коммивояжера.

1.1 Описание задачи коммивояжера

Задача коммивояжера может быть описана следующим образом - существует некоторе количество городов, каждый из которых связан с другими путями, имеющими определённую длину. Необходимо найти самый выгодный путь, начинающийся в исходном городе и проходящий ровно один раз через каждый из городов с последующим возвратом в исходный город.

Для представления задачи мы будем использовать графовую модель. Город представляется как один из узлов графа, а пути к другим городам - как рёбра. Критерий выгодности представляется как метка на ребре, обозначающая расстояние между городами, которые это ребро соединяет. Модельный граф задачи является полностью связным, то есть между любой произвольной парой различных вершин существует ребро с ненулевой меткой.

1.2 Алгоритм полного перебора

Пусть существует N городов, представленных в виде графовой модели. Для решения задачи коммивояджера методом полного перебора, нам необходимо перебрать всевозможные корректные для этой задачи пути. Данный метод даёт гарантированно идеальное решение для любого графа, однако сложность этого алгоритма не позволяет применять его для реальных задач, поскольку в следствие его рекурсивности сложность его составляет M!, что приводит к экспоненциальному росту времени выполнения программы в зависимости от количества городов.

1.3 Муравьиный алгоритм

В практических ситуациях идеальностью решения можно принебречь, если полученное решение является достаточно приближенным к нему, чтобы удовлетворять практическим запросам. Методы, которые работают, опираясь на это условие, называются эвримтическими, и решают задачу за гораздо более меньшее время, чем метод полного перебора.

В нашем случае этим методом является муравьиный метод, основанный на принципах работы реальной муравьиной колонии. Для решения задачи введём математическую модель:

Муравей является независимым агентом, который может перемещаться по графу. Муравей может выполнять следующие действия:

— оценивать длину ребра;
— оценивать количество ферромона, оставленного другими муравьями;
— запоминать посещённые города.

Введение ферромона позволяет муравьям косвенно обмениваться информацией о выбранных маршрутах.

Муравей начинает свой путь в случайно выбранном городе. Для того, чтобы перейти к следующему городу, для каждого пути, ведущего в другой город, если он не был посещён заранее, считается вероятность того, что муравей пойдёт именно по этому ребру по формуле 1.1:

$$p_{i,j} = \frac{(\tau_{i,j}^{\alpha})(\eta_{i,j}^{\beta})}{\sum(\tau_{i,j}^{\alpha})(\eta_{i,j}^{\beta})} \tag{1.1}$$

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.

При β равном нулю алгоритм работает как классический жадный алгоритм. При α равном нулю алгоритм сходится к некоторому локальному минимому. При решении задачи необходимо исследовать параметры и подобрать оптимальные на основе проведённых экспериментов.

При посещении всех городов муравей оставляет ферромон на пройденных рёбрах. Количество ферромона обратно пропорционально длине пути, который был пройдён муравьём:

$$\tau_{i,j} = (1 - \rho)\tau_{i,j} + \Delta\tau_{i,j}, \tag{1.2}$$

где $\rho_{i,j}$ - доля феромона, который испарится;
$\tau_{i,j}$ - количество феромона на дуге ij;
$\Delta\tau_{i,j}$ - количество отложенного феромона, вычисляется по формуле ??.

1.4 Вывод

В данном разделе были рассмотрены основные теоретические сведения об алгоритме полного перебора и муравьином алгоритме. В результате были сделаны выводы о том, что на вход алгоритму полносвязный граф, отражающий задачу коммивояжера, на выходе программа возвращает длину пути и сам путь. Алгоритмы работают на графах с размерностями от 2 до физически возможного предела для используемой машины. В качестве критерия для сравнения эффективности алгоритмов будет использоваться время работы на графах различного размера.

2 Конструкторский раздел

В данном разделе будут рассмотрены схемы, структуры данных, способы тестирования, описания памяти для следующих алгоритмов:

— алгоритм полного перебора;
— муравьиный алгоритм.

2.1 Тестирование алгоритмов

Описание классов эквивалентности:

1) проверка работы на корректном графе;
2) проверка работы на некорректном графе;

Описание тестов:

1) тест на общем случае - на вход подаётся граф размерама n, возвращаемый результат сравнивается с заранее известным правильным результатом;
2) тест на маленьком графе - на вход подаётся граф с количеством вершин меньше двух, возвращаемый результат является расстоянием равным 0 и пустым путём.

2.2 Алгоритм полного перебора

Используемые типы и структуры данных включают в себя:

1) integer, целое число - используется для хранения индексов массива, размера массива;
2) vector, подвид списка - используется для хранения пути и узлов;
3) bool, логическая переменная - используется в логических операциях;
4) array, массив целых чисел - используется для хранения серии целых чисел.

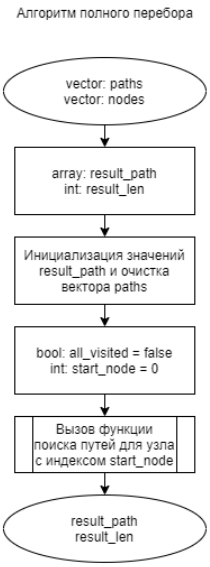

Рисунок 2.1 — Схема алгоритма полного перебора часть 1

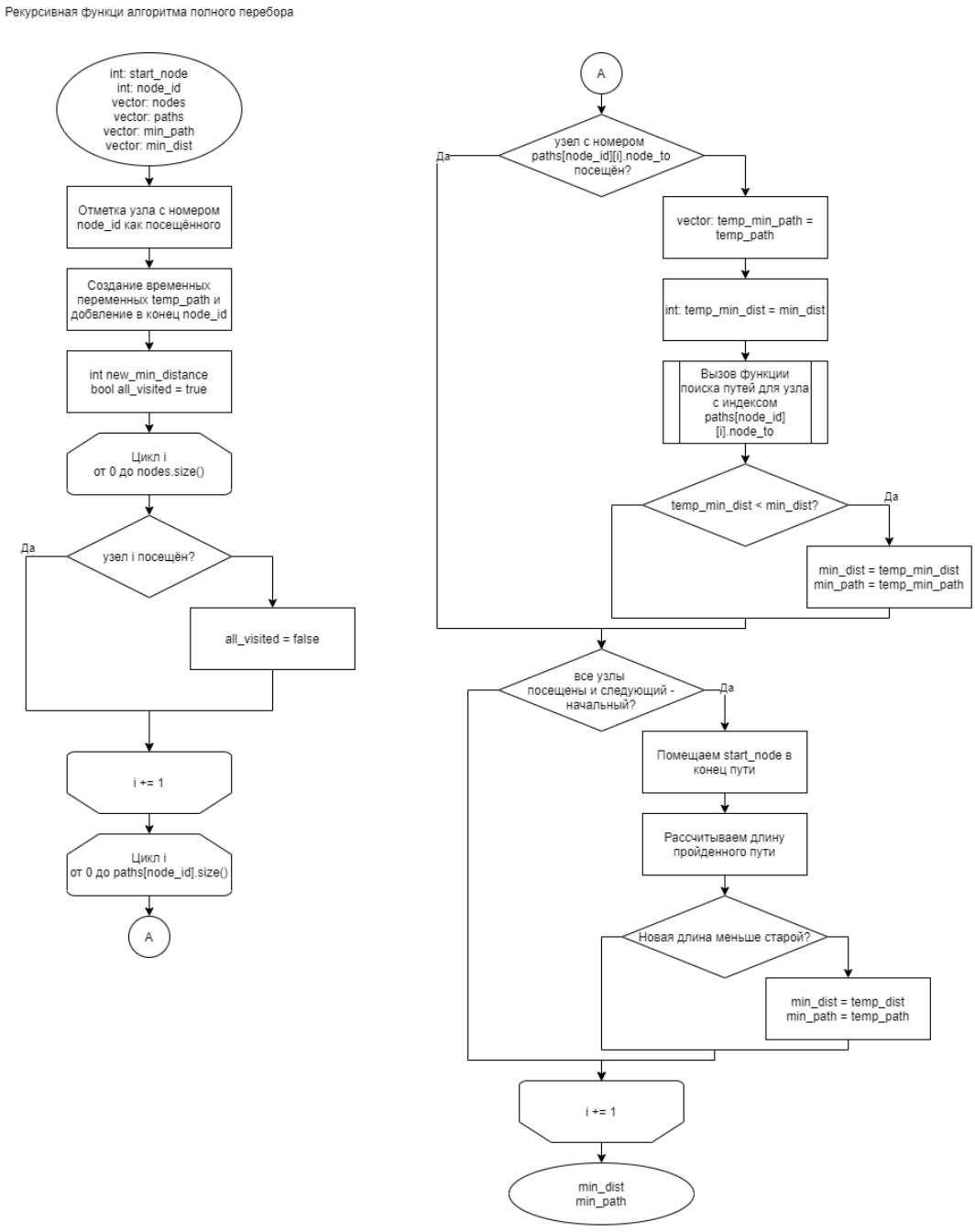

Рисунок 2.2 — Схема алгоритма полного перебора часть 2

2.3 Муравьиный алгоритм

Используемые типы и структуры данных включают в себя:

1) integer, целое число - используется для хранения индексов массива, размера массива;
2) vector, подвид списка - используется для хранения пути и узлов;
3) bool, логическая переменная - используется в логических операциях;
4) array, массив целых чисел - используется для хранения серии целых чисел.

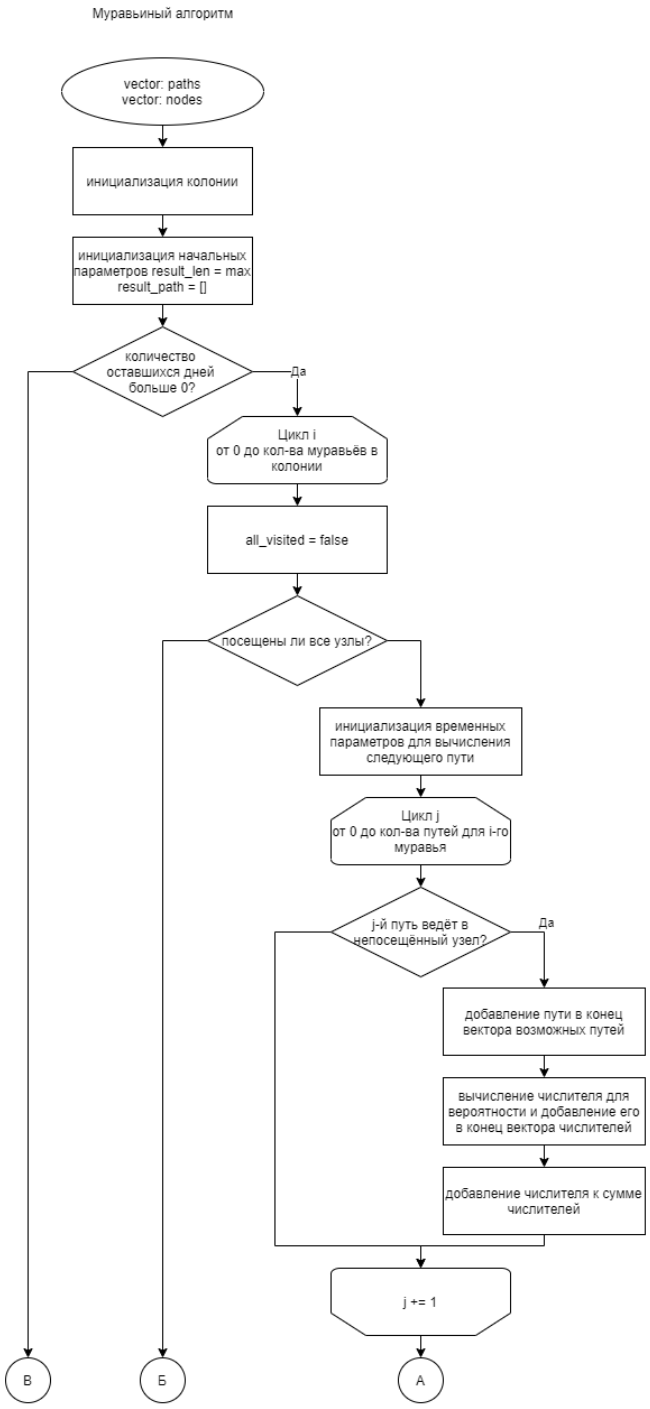

Рисунок 2.3 — Схема муравьиного алгоритма часть 1

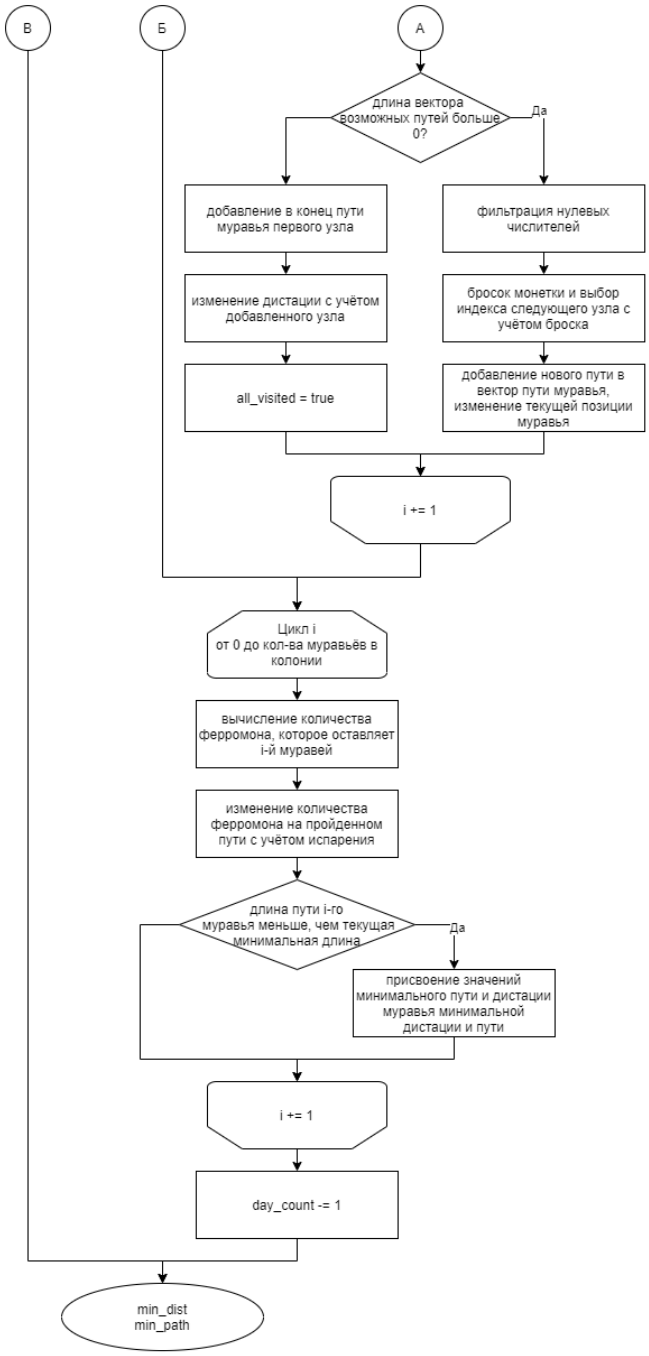

Рисунок 2.4 — Схема муравьиного алгоритма часть 2

2.4 Функциональная схема ПО

На изображении ниже представлена функиональная схема разрабатываемого ПО. На вход подаётся вектор путей и вектор узлов, при помощи алгоритмов, реализованных на языке C++ мы получаем в результате работы наименьший путь и длину наименьшего пути.

Рисунок 2.5 — IDEF0 диаграмма разрабатываемой программы

2.5 Вывод

В данном разделе были рассмотрены схемы алгоритма полного перебора и муравьиного алгоритма. Были определены тесты для каждого алгоритма и описаны типы и структуры данных, использующихся в алгоритмах. Также была приведена функциональная схема разрабатываемого ПО.

3 Технологический раздел

В данном разделе будут рассмотрены подробности реализации описаных выше алгоритмов. Также будут обоснованы выбор языка программирования для реализации, выбор библиотек для проведения экспериментов и представлены важные фрагменты кода написанной в рамках работы программы.

3.1 Выбор языка программирования

В качестве языка программирования для реализации данной лабораторной работы использовался язык программирования C++ поскольку он предоставляет широкие возможности для эффективной реализации алгоритмов. В качестве среды разработки использовалась Microsoft Visual Studio 2019 по причине того, что данная среда имеет встроенные средства отладки и анализа работы программы, позволяющие быстро и эффективно писать код.

3.2 Сведения о модулях программы

Реализованное ПО состоит из трёх модулей:

1) brute_force - в данном модуле реализован алгоритм полного перебора;
2) ants - в данном модуле реализован муравьиный алгоритм;
3) lab_6 - основной файл программы, где находится точка входа;
4) tests - реализация тестов алгоритмов;
5) time - реализация замеров времени работы программы.

3.3 Реализация алгоритмов

Листинг 3.1 — Реализация алгоритма полного перебора

```
1  void brute_force::brute_force_search(std::vector<std::vector<Path>>& path_matrix,
       std::vector<Node>& nodes, std::vector<int>& result_path, int& result_len)
2  {
3      result_len = std::numeric_limits<int>::max();
4      result_path.clear();
5      bool all_visited = false;
6      int start_node = 0;
7      find_path_for_node(start_node, 0, nodes, path_matrix, result_path, result_len);
8  }
9
10 void brute_force::find_path_for_node(int &start_node, int node_id, std::vector<Node>
       nodes, std::vector<std::vector<Path>>& paths, std::vector<int>& min_path, int&
       min_dist)
11 {
12     nodes[node_id].visited = true;
13
14     std::vector<int> temp_path = min_path;
```

```
15      temp_path.push_back(node_id);
16
17      int new_min_distance;
18      bool all_visited = true;
19      for (auto& node : nodes)
20      {
21          if (node.visited == false)
22              all_visited = false;
23      }
24
25      for (int i = 0; i < paths[node_id].size(); i++)
26      {
27          if (nodes[paths[node_id][i].node_to].visited == false)
28          {
29              std::vector<int> temp_min_path = temp_path;
30              int temp_min_dist = min_dist;
31              find_path_for_node(start_node, paths[node_id][i].node_to, nodes, paths,
                    temp_min_path, temp_min_dist);
32              if (temp_min_dist < min_dist)
33              {
34                  min_dist = temp_min_dist;
35                  min_path = temp_min_path;
36              }
37          }
38
39          if (nodes[paths[node_id][i].node_to].visited == true && all_visited &&
                paths[node_id][i].node_to == start_node)
40          {
41              temp_path.push_back(start_node);
42
43              int temp_dist = 0;
44              int from;
45              int to;
46
47              for (int i = 0; i < temp_path.size() - 1; i++)
48              {
49                  from = temp_path[i];
50                  to = temp_path[i + 1];
51                  for (auto& path : paths[from])
52                  {
53                      if (path.node_to == to)
54                          temp_dist += path.distance;
55                  }
56              }
57
58              if (temp_dist < min_dist)
59              {
```

```
                min_dist = temp_dist;
                min_path = temp_path;
            }
        }
    }
}
```

Листинг 3.2 — Реализация муравьиного алгоритма

```
void max_elem(std::vector<int> arr, int& max)
void ants::ant_search(std::vector<std::vector<Path>>& paths, std::vector<Node>&
    nodes, std::vector<int>& result_path, int& result_len)
{
    // initialize ants
    std::vector<ant> colony;

    for (int i = 0; i < ant_count; i++)
    {
        ant temp;
        temp.current_pos = rand() \% nodes.size();
        temp.distance = 0;
        temp.path.push_back(temp.current_pos);
        colony.push_back(temp);
    }

    result_len = std::numeric_limits<int>::max();
    result_path.clear();

    while (day_count > 0)
    {
        for (auto& ant : colony)
        {
            bool all_visited = false;
            while (!all_visited)
            {
                // Probability of going to node from paths
                double temp_sum = 0;
                std::vector<double> temp_top;
                std::vector<double> temp_prob;
                std::vector<Path> avalible_paths;

                for (auto& path : paths[ant.current_pos])
                {
                    if (!ant.check_node_for_visided(path.node_to))
                    {
                        avalible_paths.push_back(path);
```

```
37                          double temp = pow(1.0f / (double)path.distance, alpha) *
                                pow(path.ferromone, beta);
38                          temp_sum += temp;
39                          temp_top.push_back(temp);
40                      }
41                  }
42
43                  if (avalible_paths.size() > 0)
44                  {
45                      for (int i = 0; i < temp_top.size(); i++)
46                      {
47                          if (temp_top[i] > 0)
48                              temp_prob.push_back(temp_top[i] / temp_sum);
49                      }
50
51                      double coin = ((double)rand() / (double)RAND_MAX);
52
53                      int index = 0;
54                      while (coin >= 0)
55                      {
56                          coin -= temp_prob[index];
57                          index++;
58                          if (index == avalible_paths.size())
59                              index = 0;
60                      }
61
62                      ant.path.push_back(avalible_paths[index].node_to);
63                      ant.current_pos = avalible_paths[index].node_to;
64                      ant.distance += avalible_paths[index].distance;
65                  }
66                  else
67                  {
68                      ant.path.push_back(ant.path[0]);
69
70                      for (int i = 0; i < paths[ant.current_pos].size(); i++)
71                      {
72                          if (paths[ant.current_pos][i].node_to == ant.path[0])
73                              ant.distance += paths[ant.current_pos][i].distance;
74                      }
75
76                      all_visited = true;
77                  }
78              }
79          }
80
81          for (auto& ant_unit : colony)
82          {
```

```
83              double delta_ferr = ferr_for_ant / (double)ant_unit.distance;
84              for (int i = 0; i < ant_unit.path.size() - 1; i++)
85              {
86                  for (auto& path : paths[ant_unit.path[i]])
87                  {
88                      if (path.node_to == ant_unit.path[i + 1])
89                      {
90                          path.ferromone = (1 - evap_rate) * path.ferromone +
                                delta_ferr;
91                      }
92                  }
93              }
94
95              if (ant_unit.distance < result_len)
96              {
97                  result_len = ant_unit.distance;
98                  result_path = ant_unit.path;
99              }
100
101             ant_unit.reset_ant();
102             ant_unit.current_pos = rand() \% nodes.size();
103         }
104
105         day_count -= 1;
106     }
107 }
```

3.4 Реализация тестирования алгоритмов

Для тестирования алгоритмов было реализованы следующие тесты:

1) тест на общем случае для алгоритма полного перебора - на вход подаётся граф размерама n, возвращаемый результат сравнивается с заранее известным правильным результатом;

2) тест на слишком маленьком графе - на вход подаётся граф с количеством вершин меньше двух, возвращаемый результат является расстоянием равным 0 и пустым путём;

3) тест на неполносвязном графе - на вход подаётся граф не являющийся полносвязным, возвращаемый результат является расстоянием равным 0 и пустым путём.

Листинг 3.3 — Реализация тестов

```
1 #include "tests.h"
2
3 void test()
4 {
5     brute_force br_fc;
6     ants ants_s;
```

```
7
8       std::vector<std::vector<Path>> path_mat = {
9           {{7, 1, 1}, {8, 2, 1}, {7, 3, 1}, {3, 4, 1}},
10          {{7, 0, 1}, {6, 2, 1}, {8, 3, 1}, {4, 4, 1}},
11          {{8, 0, 1}, {6, 1, 1}, {4, 3, 1}, {7, 4, 1}},
12          {{7, 0, 1}, {8, 1, 1}, {4, 2, 1}, {7, 4, 1}},
13          {{3, 0, 1}, {4, 1, 1}, {7, 2, 1}, {7, 3, 1}}
14      };
15
16      std::vector<Node> nodes = {{1, 7, 0}, {4, 0, 0}, {9, 4, 0}, {8, 8, 0}, {2, 4,
            0}};
17
18      std::vector<int> result_path;
19      int result_len = 0;
20
21      // common data test for brute force
22      br_fc.brute_force_search(path_mat, nodes, result_path, result_len);
23      std::vector<int> true_result_path = { 0, 3, 2, 1, 4, 0 };
24      int true_result_len = 24;
25      std::cout << "Test on common data for brute force: ";
26      if (true_result_len == result_len && result_path == true_result_path)
27      {
28          std::cout << "PASSED" << std::endl;
29      }
30      else
31      {
32          std::cout << "NOT PASSED" << std::endl;
33      }
34
35      // small graph data test for brute force
36
37      path_mat.clear();
38      nodes = { {1, 7, 0} };
39      result_path.clear();
40      result_len = 0;
41
42      br_fc.brute_force_search(path_mat, nodes, result_path, result_len);
43
44      true_result_len = 0;
45      true_result_path.clear();
46
47      std::cout << "Test on small graph data for brute force: ";
48      if (true_result_len == result_len && result_path == true_result_path)
49      {
50          std::cout << "PASSED" << std::endl;
51      }
52      else
```

```
53      {
54          std::cout << "NOT PASSED" << std::endl;
55      }
56
57      // small graph data test for ants
58
59      path_mat.clear();
60      nodes = { {1, 7, 0} };
61      result_path.clear();
62      result_len = 0;
63
64      ants_s.ant_search(path_mat, nodes, result_path, result_len);
65
66      true_result_len = 0;
67      true_result_path.clear();
68
69      std::cout << "Test on small graph data for ant algorythm: ";
70      if (true_result_len == result_len && result_path == true_result_path)
71      {
72          std::cout << "PASSED" << std::endl;
73      }
74      else
75      {
76          std::cout << "NOT PASSED" << std::endl;
77      }
78
79      // incorrect graph data test for brute force
80
81      path_mat = {
82          {{7, 1, 1}, {8, 2, 1}, {7, 3, 1}, {3, 4, 1}},
83          {{7, 0, 1}, {6, 2, 1}, {8, 3, 1}, {4, 4, 1}},
84          {{8, 0, 1}, {6, 1, 1}, {4, 3, 1}, {7, 4, 1}},
85          {{7, 0, 1}, {8, 1, 1}, {4, 2, 1}, {7, 4, 1}},
86          {{3, 0, 1}, {4, 1, 1}, {7, 2, 1}}
87      };
88
89      nodes = { {1, 7, 0}, {4, 0, 0}, {9, 4, 0}, {8, 8, 0}, {2, 4, 0} };
90
91      result_path.clear();
92      result_len = 0;
93
94      br_fc.brute_force_search(path_mat, nodes, result_path, result_len);
95
96      true_result_len = 0;
97      true_result_path.clear();
98
99      std::cout << "Test on incorrect graph data for brute force: ";
```

```
100     if (true_result_len == result_len && result_path == true_result_path)
101     {
102         std::cout << "PASSED" << std::endl;
103     }
104     else
105     {
106         std::cout << "NOT PASSED" << std::endl;
107     }
108 
109     // incorrect graph data test for ants
110 
111     path_mat = {
112         {{7, 1, 1}, {8, 2, 1}, {7, 3, 1}, {3, 4, 1}},
113         {{7, 0, 1}, {6, 2, 1}, {8, 3, 1}, {4, 4, 1}},
114         {{8, 0, 1}, {6, 1, 1}, {4, 3, 1}, {7, 4, 1}},
115         {{7, 0, 1}, {8, 1, 1}, {4, 2, 1}, {7, 4, 1}},
116         {{3, 0, 1}, {4, 1, 1}, {7, 2, 1}}
117     };
118 
119     nodes = { {1, 7, 0}, {4, 0, 0}, {9, 4, 0}, {8, 8, 0}, {2, 4, 0} };
120 
121     result_path.clear();
122     result_len = 0;
123 
124     ants_s.ant_search(path_mat, nodes, result_path, result_len);
125 
126     true_result_len = 0;
127     true_result_path.clear();
128 
129     std::cout << "Test on incorrect graph data for ant algorythm: ";
130     if (true_result_len == result_len && result_path == true_result_path)
131     {
132         std::cout << "PASSED" << std::endl;
133     }
134     else
135     {
136         std::cout << "NOT PASSED" << std::endl;
137     }
138 }
```

3.5 Вывод

В данной разделе были представлены реализации алгоритма полного перебора и муравьиного алгоритма, а также представлена реализация модуля тестирования реализованных алгоритмов.

4 Экспериментальный раздел

В данном разделе описывается подбор параметров муравьиного алгоритма и измеренте временных характеристики алгоритмов полного перебра и муравьиного алгоритма, а также делается вывод об эффективности алгоритмов.

4.1 Технические характеристики

— Операционная система - Windows 10, 64-bit;
— Оперативная память - 16 GiB;
— Процессор - Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz, 6 ядер, 12 потоков.

4.2 Результаты подбора параметров

В муравьином алгоритме вычисления производятся на основе нескольких настроиваих параметров. По причине этого нам необходимо подобрать лучшую комбинацию параметров, при которой алгоритм работает быстрее и лучше. Эксперимент проводится на графе имеющем 10 узлов с расстояниями от 1 до 10. Такой размер выбран по причине того, что 10 узлов могут быть обработаны алгоритмом полного перебора за приемлемый отрезок времени. Параметры перебора: количество муравьёв - 5, количество дней - от 50 до 500 с шагом 50, количество ферромона, которое переносит один муравей - от 10 до 50 с шагом 10, альфа - от 0 до 1 с шагом 0.2, уровень испарения ферромонов от 0 до 1 с шагом 0.2. Так как количество вариаций превышает несколько тысяч, в таблицы добавлялись только результаты с путём, 80% которого меньше идеального пути, полученного с помощью алгоритмов полного перебора.

Обозначения:
iters - кол-во итераций.
ants - кол-во муравьёв.
f - кол-во ферромона для одного муравья.
min - минимальная дистанция для данного набора параметров.
true min - истинная минимальная дистанция, найденная полным перебором.

Набор параметров, который позволил найти наилучший путь:
количество итераций - 300;
количество муравьёв - 5;
альфа - 0.6;
бета - 0.4;
уровень испарения - 0.8;
количество ферромона для муравья - 10.

Таблица 4.1 — Перебор параметров часть 1

iters	ants	α	β	p	f	min	true min
150	5	0	1	0	10	34.6	28
150	5	0	1	0.2	40	33.8	28
200	5	0.4	0.6	0.4	20	34.8	28
250	5	0	1	0.4	40	34.2	28
250	5	0	1	0.8	10	34.8	28
250	5	0	1	0.8	40	33.8	28
250	5	0.2	0.8	0.8	20	34.6	28
300	5	0	1	0	10	34	28
300	5	0	1	0.2	10	34.6	28
300	5	0	1	0.4	10	33.8	28
300	5	0	1	0.6	10	34.8	28
300	5	0.4	0.6	0.4	10	34.8	28
300	5	0.6	0.4	0.6	40	34.4	28
300	5	0.6	0.4	0.8	10	33.6	28

Таблица 4.2 — Перебор параметров часть 2

iters	ants	α	β	p	f	min	true min
350	5	0	1	0	20	34.4	28
350	5	0	1	0	30	33.8	28
350	5	0	1	0.8	10	33.8	28
350	5	0	1	0.8	30	34.8	28
350	5	0.2	0.8	0.2	20	34.8	28
350	5	0.2	0.8	0.4	20	34.8	28
350	5	0.2	0.8	0.8	40	34.8	28
350	5	0.4	0.6	0.8	20	34.8	28
350	5	0.6	0.4	0	30	34.6	28
400	5	0	1	0	20	34.6	28
400	5	0	1	0	30	34.6	28
400	5	0	1	0.4	20	34.2	28
400	5	0	1	0.8	20	34.8	28
400	5	0	1	0.8	30	34.4	28
400	5	0.2	0.8	0	30	34.2	28
400	5	0.2	0.8	0.6	30	34.8	28
400	5	0.2	0.8	0.6	40	34	28
400	5	0.2	0.8	0.8	30	34.4	28
400	5	0.4	0.6	0.4	10	34.8	28
400	5	0.4	0.6	0.6	10	34.4	28
400	5	0.6	0.4	0	10	34.8	28

Таблица 4.3 — Перебор параметров часть 3

iters	ants	α	β	p	f	min	true min
450	5	0	1	0	30	34.6	28
450	5	0	1	0.2	10	34.4	28
450	5	0	1	0.4	20	34.4	28
450	5	0	1	0.4	30	34	28
450	5	0	1	0.4	40	34.6	28
450	5	0	1	0.8	10	34.4	28
450	5	0	1	0.8	20	34.2	28
450	5	0.2	0.8	0.6	10	34	28
450	5	0.2	0.8	0.6	40	34.8	28
450	5	0.4	0.6	0	20	34.8	28
450	5	0.4	0.6	0.8	30	34.2	28
450	5	0.6	0.4	0	20	34.6	28
450	5	0.6	0.4	0.4	10	34.8	28
450	5	0.8	0.2	0.2	30	34.8	28
450	5	0.8	0.2	0.4	10	34.8	28
500	5	0	1	0	30	34.2	28
500	5	0	1	0.4	40	34.8	28
500	5	0	1	0.6	20	34.4	28
500	5	0	1	0.6	30	34.8	28
500	5	0	1	0.8	10	34.8	28
500	5	0	1	0.8	30	34.4	28
500	5	0	1	0.8	40	34.4	28
500	5	0.2	0.8	0.2	10	34.6	28
500	5	0.2	0.8	0.4	40	34.6	28
500	5	0.2	0.8	0.8	20	34.8	28
500	5	0.2	0.8	0.8	30	34	28
500	5	0.4	0.6	0.6	40	34.6	28
500	5	0.4	0.6	0.8	10	34.2	28
500	5	0.6	0.4	0.4	30	34.8	28
500	5	0.6	0.4	0.6	40	33.8	28

4.3 Результаты экспериментов

Таблица 4.4 — Время работы алгоритмов

кол-во узлов	t полного перебора (нс)	t муравьиного алгоритма (нс)
2	41140	50990360
3	95460	123472630
4	209590	128274640
5	692210	137194230
6	4723110	312229910
7	41685220	458251240
8	212235570	418253070
9	1672092450	495543340
10	14021435680	509634800

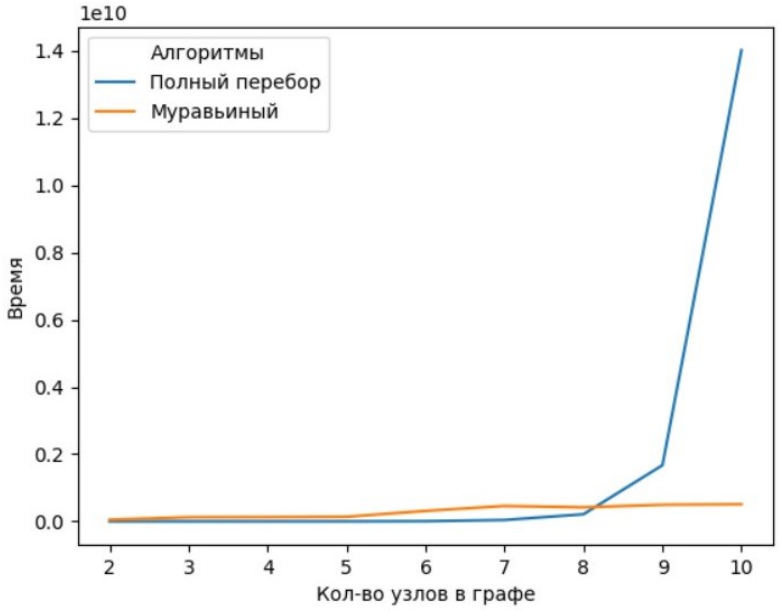

Рисунок 4.1 — График зависимости времени работы от размерности массивов

4.4 Вывод

В результате эксперимента было получено, что оптимальными параметрами для муравьиного алгоритма являются количество итераций - 300, количество муравьёв - 5, альфа - 0.6, бета - 0.4, уровень испарения - 0.8, количество ферромона для муравья - 10. При сравнении времени выполнения при достижении 10 узлов муравьиный алгоритм работает более чем в 27 раз быстрее. В результате можно сделать вывод о том, что использование муравьиного алгоритма позволяет решать задачу коммивояжера значительно быстрее, чем алгоритм полного перебора.

Заключение

В процессе выполнения данной лабораторной работы были изучены алгоритм полного перебора и муравьиный алгоритм. Были выполнены анализ алгоритмов и представлены схемы алгоритмов, а также функциональная схема ПО. После чего эти алгоритмы были реализованы при помощи языка C++ в IDE Visual Studio 2019. Помимо этого были произведены эксперименты с целью получить информацию о временной производительности алгоритмов. В результате эксперимента было получено, что оптимальными параметрами для муравьиного алгоритма являются количество итераций - 300, количество муравьёв - 5, альфа - 0.6, бета - 0.4, уровень испарения - 0.8, количество ферромона для муравья - 10. При сравнении времени выполнения при достижении 10 узлов муравьиный алгоритм работает более чем в 27 раз быстрее. В результате можно сделать вывод о том, что использование муравьиного алгоритма позволяет решать задачу коммивояжера значительно быстрее, чем алгоритм полного перебора. Целью данной лабораторной работы являлось изучение алгоритмов полного перебора и муравьиного алгоритма на примере задачи коммивояжера, что было успешно достигнуто.

Список литературы

[1] Мудров В.И. Задача о коммивояжере. — М.: Наука, 1969. — 62 с.

[2] Дориго М. Муравьиная система. — М.: Наука, 1996. — 29 с. [3] "Документация по языку C++"[электронный ресурс]. Режим доступа: https://docs.microsoft.com/ru-ru/cpp/cpp/?view=msvc-16 свободный (Дата обращения 1.12.21)

[4] Процессор Intel® Core™ i7-9750H (12 МБ кэш-памяти, до 4,50 ГГц) [электронный ресурс] https://www.intel.ru/content/www/ru/ru/products/sku/191045/intel-core-i79750h-processor-12m-cache- свободный (Дата обращения: 2.12.21)

[5] <chrono> [электронный ресурс] https://docs.microsoft.com/ru-ru/cpp/standard-library/chrono?view= свободный (Дата обращения: 5.12.21)