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 по дисциплине "Анализ алгоритмов"

Тема Муравьиный алгоритм
Студент Федоров В.П.
Группа ИУ7-53Б
Преподаватели Волкова Л.Л., Строганов Ю.В.

Москва — 2021 г.

Оглавление

Введение

Целью данной лабораторной работы является приобритение навыка параметризации метода на примере решения задачи коммивояжера методом, основанным на муравьином алгоритме.

Задачи лабораторной работы:

1. параметризация - выбор таких параметров или их сочетаний метода, при которых этот метод решает класс задач с наилучшим качеством;

2. провести сравнительный анализ эвристического метода и метода полного перебора;

Аналитическая часть

Задача коммивояжёра (или TSP от англ. Travelling salesman problem) — 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.

Сама задача формулируется довольно просто: имеется n городов, расстояния между которыми нам известны. Коммивояжер находится в одном из этих городов, и желает объехать все города, при этом посетив каждый из них ровно один раз, а затем вернуться в исходный город. Кроме того, искомый путь должен иметь наименьшую длину.

Расстояния между городами обычно задаются с помощью матрицы смежности. Задача коммивояжера является NP-трудной, даже когда матрица расстояний между городами симметрична, то есть задача коммивояжера не может быть проверена за полиномиальное время, однако задача разрешения для задачи коммивояжера принадлежит к классу NP-полных задач.

В данной лабораторной работе рассматривается случай, когда из любого города можно попасть в любой город и все дороги(маршруты) являются неориентированными (то есть рассматривается полный граф и симметричная матрица смежности).

Алгоритмы решения задачи коммивояжёра

Существует множество различных способов решения этой задачи, как точных, так и приближенных. В качестве примера точного метода ее решения можно привести полный перебор — перебор всевозможных вариантов и в выбор среди них оптимального. Все алгоритмы, сокращающие полный перебор, не дают точного значения. С их помощью можно найти только приближённое решение задачи, при этом не гарантируется, что погрешность вычислений будет низка.

Полный перебор

Алгоритм полного перебора (также известен как метод грубой силы) осуществляет поиск всех решений путём перебора всех вариантов в количестве N! штук, позволяя получить точное решение задачи. Однако как уже было сказано, задача коммивояжёра относится к классу NP-трудных: 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

В основе муравьиного алгоритма лежат принципы самоорганизации муравьиной колонии.

Идея муравьиного алгоритма заключается в моделировании поведения муравьев, связанного с их способностью находить кратчайший путь от муравейника к источнику пищи и адаптироваться к изменяющимся условиям, находя новый кратчайший путь. При своем движении муравей метит свой путь феромоном, и эта информация используется другими муравьями. Это и является главным правилом поведения каждого из представителей колониив случае, если старый маршрут стал недоступен.

Рассмотрим случай, когда на оптимальном пути возникает преграда. В этом случае необходимо определение нового оптимального пути. Дойдя до преграды, муравьи с равной вероятностью будут обходить её справа и слева. То же самое будет происходить и на обратной стороне преграды. Однако, те муравьи, которые случайно выберут кратчайший путь, будут быстрее его проходить, и за несколько передвижений он будет более обогащён феромоном. Поскольку движение муравьёв определяется концентрацией феромона, то следующие будут предпочитать именно этот путь, продолжая обогащать его феромоном до тех пор, пока этот путь по какой-либо причине не станет недоступен.

Испарение феромона в воздухе гарантирует, что найденное решение не будет единственным, и муравьи будут продолжать искать и другие пути. Если мы моделируем процесс такого поведения на графе, ребра которого — всевозможные пути движения, то в течение некоторого времени наиболее обогащенный феромоном путь и будет наикратчайшим, что и даст решение задачи.

Преимуществами алгоритма являются невысокая погрешность найденного решения, низкие временные затраты при работе с графами большой размерности, модифицируемость алгоритма и возможность распараллеливания.

Применение муравьиного алгоритма

Любой муравьиный алгоритм, независимо от модификаций, представим вследующем виде.

1. Инициализация параметров.

Пока (условия выхода не выполнены):

1. создаём Муравьёв;

2. поиск решений;

3. обновление феромона.

Пусть имеется M городов и задана симметричная матрица смежности $D[MxM]$. Рассмотрим каждый шаг в цикле более подробно.

Инициализация параметров

На этом этапе задаётся начальный уровень феромона. Ониницализируется небольшим положительным числом для того, чтобы наначальном шаге вероятности перехода в следующую вершину не былинулевыми. Кроме начального уровня феромонов необходимо также задать следующие параметры.

1. α - параметр влияния длины пути.
2. β - параметр влияния феромона ($\alpha + \beta = \text{const}$).
3. $\rho \in (0, 1)$ - коэффицент испарения феромона.
4. t_{max} - количество поколений муравьёв (количество итераций в алгоритме).
5. τ_{min} - минимальное возможное значение феромона. Феромон не должен падать ниже этой константы и обнуляться, чтобы не обнулялась вероятность (см. ниже).
6. τ_{start} - начальное значение феромона.
7. Q - количество феромона, переносимого муравьем.

Создание муравьёв

В каждый город помещается по муравью (количество муравьев равно количеству городов).

Поиск решений

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.

Пусть k-й муравей находится в i-м городе. Тогда вероятность посещения города j, который ранее на данной итерации муравьем не посещался, будет вычисляться по следующей формуле.

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

где

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

$\eta_{i,j} = \dfrac{1}{D_{ij}}$ – привлекательность города для муравья;

α – параметр влияния длины пути;

β – параметр влияния феромона.

Очевидно, что при $\beta = 0$ алгоритм превращается в классический жадный алгоритм. Выбор правильного соотношения параметров является предметом исследований, и в общем случае производится на основании опыта.

Обновление феромона

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

$$\tau_{ij}(t+1) = (1-\rho)\tau_{ij}(t) + \sum_{k=1}^{n} \Delta\tau_{k,ij}(t), \tag{2}$$

где

ρ_{ij} – коэффицент испарения феромона ($\rho \in (0, 1)$);

τ_{ij} – количество феромона на дуге ij;

$\Delta\tau_{ij}(t)$ – суммарное количество феромона отложенного на ребре.

Количество феромона отложенного муравьем на данном шаге по времени, обычно определяется следующим образом.

$$\Delta\tau_{k,ij}(t) = \begin{cases} \frac{Q}{L_k}(t) & \text{, если k-ый мурваей прошел по ребру ij} \\ 0 & \text{, иначе} \end{cases} \tag{3}$$

где

Q – количество феромона, переносимого муравьем;

$L_k(t)$ – длина сформ. маршрута k-го муравья на данном шаге по времени.

Конструкторская часть

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

Требования к программе

Требования к вводу:
На вход программе подается файл с содержанием следующего формата.

1. В первой строке записано количетство городов (M).
2. На каждой из последующих строк записаны строки матрицы смежности размерности M x M.
3. Матрица симметричная.
4. На главной диагонали матрицы записаны нули.
5. Матрица смежности содержит целочисленные значения.
6. Никаких других лишниих символов в файле содержаться не должно.

Требования 2-4 являются следствием ограничений, накладываемых на граф. Эти ограничения обсуждались в аналитической части.

Требования к программе:
Программа должна решать задачу коммивояжёра и выводить длину оптимального (в случае муравьиного алгоритма) или самого короткого (в случае полного перебора) маршрута.

Схемы алгоритмов

Схема алгоритма полного перебора представлена на рисунке 1. Схема метода createAdjacencyList(), используемого в алгоритме полного перебора, представлена на рисунке 2. Метод createAdjacencyList() из исходной матрицы смежности строит список смежжности. Это делается для того, чтобы можно было более удобно реализовать полный перебор. Схема метода allPaths(), используемого в алгоритме полного перебора, представлена на рисунках 3-4. Схема метода isIn(), используемого в методе allPaths(), приведена на рисунке 5.

Обратим внимание, что в методах используется контейнер vector из стандартной библиотеки C++ и его методы. За более подробной информацией об этом контейнере и его методах следует обратиться к документации.

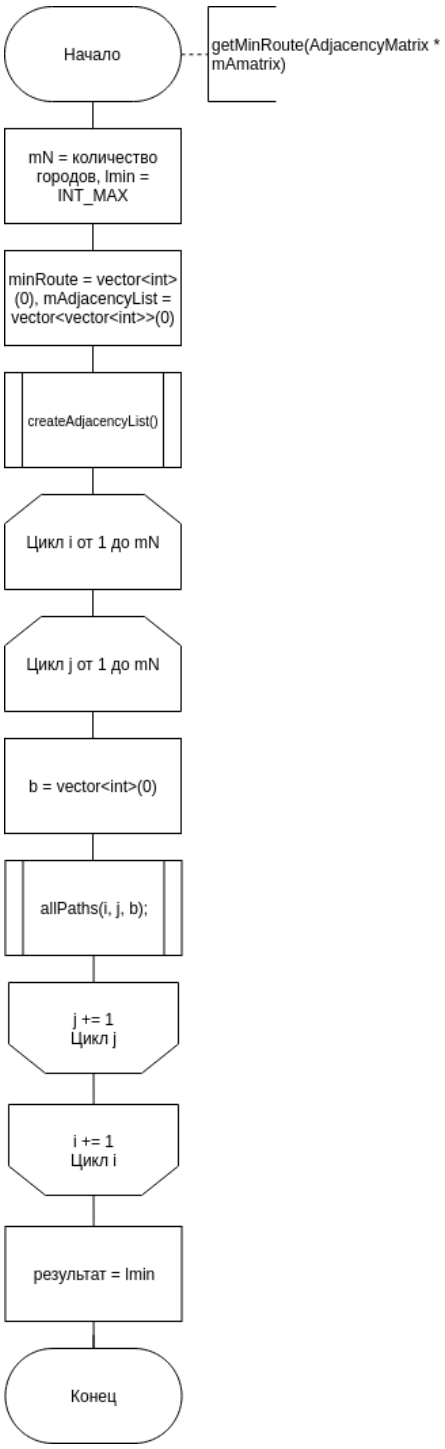

Рис. 1: Схема алгоритма полного перебора

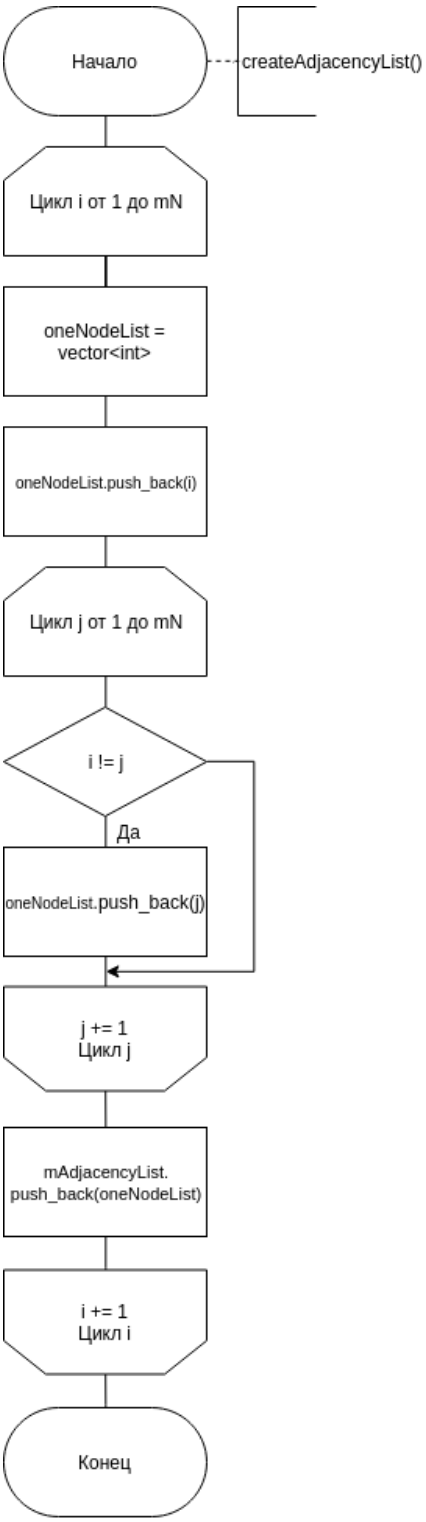

Рис. 2: Схема метода createAdjacencyList()

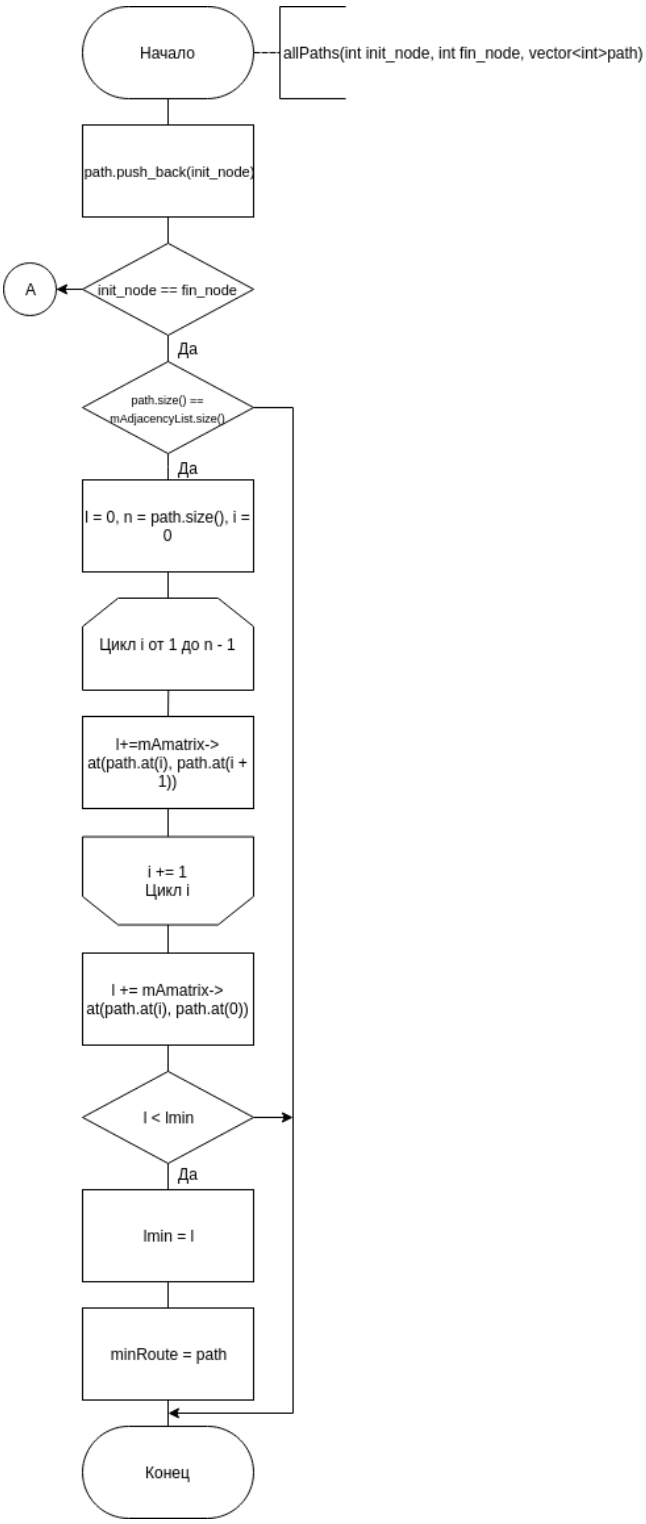

Рис. 3: Схема метода allPaths()

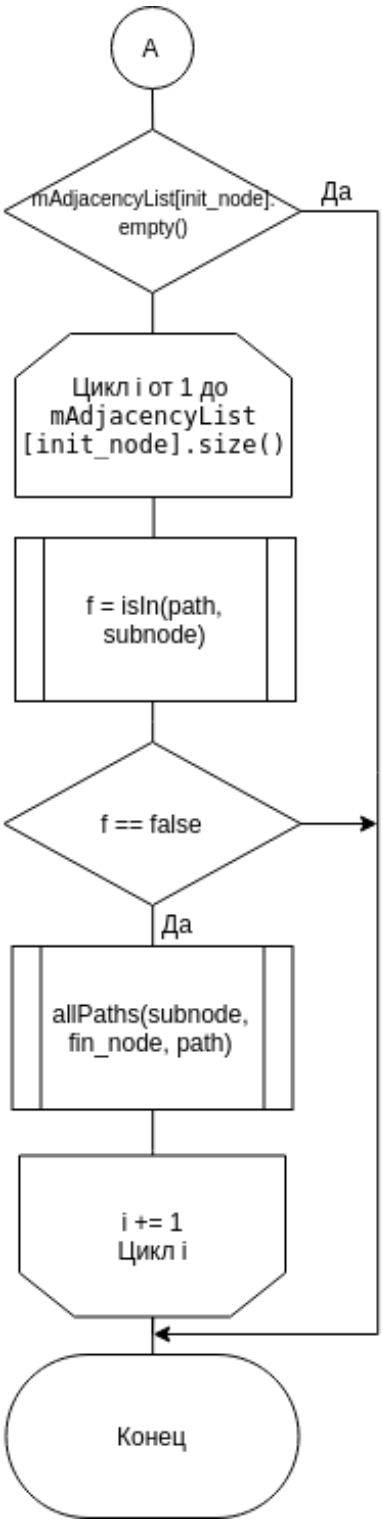

Рис. 4: Схема метода allPaths() (продолжение)

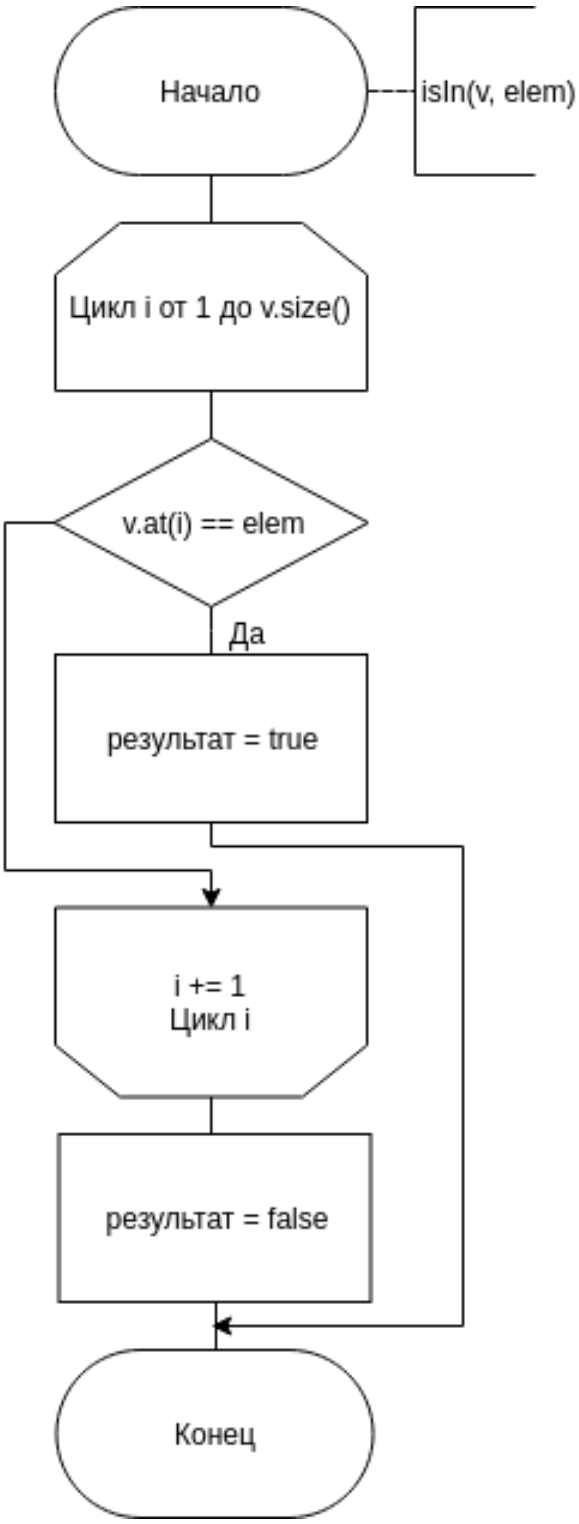

Рис. 5: Схема метода isIn()

Приведем псевдокод муравьиного алгоритма для задачи коммивояжёра [1].

1. Ввод матрицы расстояний D.
2. Инициализация параметров алгоритма — α, β, ρ, Q.
3. Инициализация ребер — присвоение начальной концентрации феромона τ_{start}.
4. Размещение муравьев в случайно выбранные города.
5. Выбор начального кратчайшего маршрута T_{best} и определение его длины L_{best}.
6. Цикл по времени жизни колонии $t = 1, ..., t_{max}$.
7. Цикл по всем муравьям $k = 1, ..., m$
8. Построить маршрут $T_k(t)$ по правилу (1) и рассчитать длину $L_k(t)$.
9. Конец цикла по муравьям.
10. Проверка всех $L_k(t)$ на лучшее решение по сравнению с L_{best}.
11. Если да, то обновить L_{best} и T_{best}.
12. Цикл по всем ребрам графа.
13. Обновить следы феромона на ребре по правилам (2) и (3).
14. Конец цикла по ребрам.
15. Конец цикла по времени.
16. Вывести кратчайший маршрут T_{best} и его длину L_{best}.

Все формулы, необходимые для муравьиного алгоритма, были в аналитическом разделе.

Технологическая часть

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

Выбор языка программирования

В качестве языка программирования для реализации программы был выбран язык C++ и фреймворк Qt, потому что:

- язык C++ имеет высокую вычислительную производительность;
- язык C++ поддерживает различные стили программирования;
- в Qt существует удобный инструмент для тестирования - QtTest - который позволяет собирать тесты в группы, собирать результаты выполнения тестов, а также уменьшить дублирование кода при схожих объектах тестирования.

Для замеров времени использовалась функция clock() модуля ctime. Эта функция возвращает количество временных тактов, прошедших с начала запуска программы. С помощью макроса CLOCKS_PER_SEC можно узнать количество пройденных тактов за 1 секунду.

Сведения о модулях программы

Программа состоит из следующих файлов:

- adjacencymatrix.h, adjacencymatrix.cpp - заголовочный файл и файл, в котором расположена реализация класса матрицы смежности;
- main.cpp - главный файл программы;
- algorithm.h, algorithm.cpp - файл и заголовочный файл, в котором расположена реализация класса Algorithm, который является родителем для классов BruteForce и AntColony;
- antcolony.h, antcolony.cpp - файл и заголовочный файл, в котором расположена реализация муравьиного алгоритма;

- bruteforce.h, bruteforce.cpp - файл и заголовочный файл, в котором расположена реализация алгоритма полгого перебора для задачи коммивояжёра;
- testsalesman.h, testsalesman.cpp - файл и заголовочный файл, в котором расположены тесты.

Листинги кода алгоритмов

В листинге 1 приведено объявление класса AntColony. В листинге 2 приведена реализация класса AntColony.

Листинг 1: Объявление класса AntColony

```
class AntColony : public Algorithm
{
public:
    AntColony();
    int getMinRoute(AdjacencyMatrix *amatrix, size_t alpha, size_t
       beta, double q, size_t tmax);
    vector<int> getMinRoute();
private:
    int lmin = INT_MAX;
    vector<int> minRoute;
    AdjacencyMatrix *mAmatrix;
    unsigned int mN;

    void getRoute(vector<int> all, int start, vector<int> &route,
       size_t &len,
                  vector<vector<double>> tao, vector<vector<double>>
                     attraction,
                  size_t alpha, size_t beta);
    vector<int> deleteFromVector(vector<int> to, int last);
    vector<double> getProbability(int from, vector<int> to, vector<
       vector<double>> tao, vector<vector<double>> attraction,
                                  size_t alpha, size_t beta);
    bool includes(int a, int b, vector<int> route);
};
```

Листинг 2: Реализация класса AntColony

```
#include "antcolony.h"

AntColony::AntColony()
    : Algorithm ()
{

}

void AntColony::getRoute(vector<int> all, int start, vector<int> &
   route, size_t &len, vector<vector<double>> tao, vector<vector<
   double>> attraction, size_t alpha, size_t beta)
{
    route.resize(0);
```

```
12      route.push_back(start);
13      vector<int> to = deleteFromVector(all, start);
14      size_t n_1 = tao.size() - 2;
15      int from;
16      double coin, sum;
17      bool flag;
18
19      for (size_t i = 0; i < n_1; i++){
20          sum = 0;
21          flag = true;
22          from = route[i];
23          vector<double> p = getProbability(from, to, tao, attraction,
                alpha, beta);
24          coin = double(rand() % 10000) / 10000;
25          for (size_t j = 0; j < p.size() && flag; j++)
26                  {
27              sum += p[j];
28              if (coin < sum)
29                          {
30                  route.push_back(to[j]);
31                  len += mAmatrix->at(from, to[j]);
32                  to = deleteFromVector(to, to[j]);
33                  flag = false;
34              }
35          }
36      }
37      len += mAmatrix->at(route[route.size()-1], to[0]);
38      route.push_back(to[0]);
39      len += mAmatrix->at(route[route.size()-1], route[0]);
40      route.push_back(route[0]);
41  }
42
43  int AntColony::getMinRoute(AdjacencyMatrix *amatrix, size_t alpha,
        size_t beta, double q, size_t tmax)
44  {
45      lmin = INT_MAX;
46      minRoute.clear();
47
48      this->mAmatrix = amatrix;
49      this->mN = static_cast<unsigned int>(mAmatrix->cities());
50
51      double tao_min, tao_start, Q;
52      vector<int> all(mN);
53      Q = 350;
54      tao_min = 0.001;
55      tao_start = 1.0 / mN;
56
57      vector<vector<int>> routes(mN);
58      vector<size_t> lens(mN);
59
60      vector<vector<double>> attraction(mN);
61      vector<vector<double>> tao(mN);
62
63      for (size_t i = 0; i < mN; i++)
```

```
64      {
65          attraction[i].resize(mN);
66          tao[i].resize(mN);
67          lens[i] = 0;
68          all[i] = static_cast<int>(i);
69          for (size_t j = 0; j < mN; j++)
70          {
71              if (i != j)
72                              {
73                  attraction[i][j] = 1.0 / mAmatrix->at(i, j);
74                  tao[i][j] = tao_start;
75              }
76          }
77      }
78
79      for (size_t time = 0; time < tmax; time++)
80          {
81          for (size_t k = 0; k < mN; k++)
82                  {
83              getRoute(all, static_cast<int>(k), routes[k], lens[k], tao
                    , attraction, alpha, beta);
84              if (lens[k] < lmin)
85              {
86                  lmin = lens[k];
87                  minRoute = routes[k];
88               }
89          }
90          for (size_t i = 0; i < mN; i++)
91              for (size_t j = 0; j < mN; j++)
92              {
93                  double sum = 0;
94                  for (size_t m = 0; m < mN; m++)
95                  {
96                      if (includes(static_cast<int>(i), static_cast<int
                            >(j), routes[m]))
97                          sum += Q/lens[m];
98                  }
99
100                 tao[i][j] = tao[i][j]*(1 - q) + sum;
101                 if (tao[i][j] < tao_min)
102                      tao[i][j] = tao_min;
103             }
104     }
105     return lmin;
106 }
107
108 vector<int> AntColony::getMinRoute()
109 {
110     return minRoute;
111 }
112
113 bool AntColony::includes(int a, int b, vector<int> route)
114 {
115     bool res = false;
```

```
116     size_t m = route.size()-1;
117     for (size_t i = 0; i < m; i++)
118         {
119         if (a == route[i] && b == route[i+1])
120             res = true;
121     }
122     return res;
123 }
124
125 vector<int> AntColony::deleteFromVector(vector<int> to, int last)
126 {
127     size_t n = to.size();
128     vector<int> cur;
129     for (size_t i = 0; i < n; i++)
130         if (to[i] != last)
131             cur.push_back(to[i]);
132     return cur;
133 }
134
135 vector<double> AntColony::getProbability(int from, vector<int> to,
136                                          vector<vector<double>> tao,
                                                vector<vector<double>>
                                                attraction,
137                                          size_t alpha, size_t beta)
138 {
139     double znam = 0, chisl = 0;
140     size_t n = to.size();
141     vector<double> result(n);
142     for (size_t i = 0; i < n; i++)
143         {
144         znam += pow(tao[from][to[i]], alpha) * pow(attraction[from][to
                [i]], beta);
145     }
146     for (size_t j = 0; j < n; j++)
147         {
148         chisl = pow(tao[from][to[j]], alpha) * pow(attraction[from][to
                [j]], beta);
149         result[j] = chisl / znam;
150     }
151     return result;
152 }
```

Листинг 3: Объявление класса BruteForce

```
1  class BruteForce : public Algorithm
2  {
3  public:
4      BruteForce();
5      int getMinRoute(AdjacencyMatrix *amatrix);
6      vector<int> getMinRouteVec();
7  private:
8      void createAdjacencyList();
9      void printVec(vector<int> &vec);
10     bool isIn(vector<int> v, int elem);
```

```
11      void allPaths(int init_node, int fin_node,
12                    vector<int> path);
13
14      int lmin = INT_MAX;
15      vector<int> minRoute;
16      AdjacencyMatrix *mAmatrix;
17      vector<vector<int>> mAdjacencyList;
18      int mN;
19  };
```

Листинг 4: Реализация класса BruteForce

```
1
2  #include "bruteforce.h"
3
4  BruteForce::BruteForce()
5      : Algorithm ()
6  {
7
8  }
9
10
11 int BruteForce::getMinRoute(AdjacencyMatrix *amatrix)
12 {
13     if (amatrix->cities() > 15)
14         throw logic_error("Search will take too much time!\n");
15
16     this->mAmatrix = amatrix;
17     this->mN = mAmatrix->cities();
18     minRoute.clear();
19
20     createAdjacencyList();
21
22     for (int i = 0; i < mN; ++i)
23         for (int j = 0; j < mN; ++j)
24         {
25             if (i != j)
26             {
27                 vector<int> b;
28                 allPaths(i, j, b);
29             }
30         }
31
32 //     printVec(minRoute);
33 //     cout << lmin << endl;
34
35     return lmin;
36 }
37
38 void BruteForce::createAdjacencyList()
39 {
40     for (int i = 0; i < mN; ++i)
41     {
42         vector<int> oneNodeList;
```

```
43          oneNodeList.push_back(i);
44
45          for (int j = 0; j < mN; ++j)
46          {
47              if (i != j)
48                  oneNodeList.push_back(j);
49          }
50
51          mAdjacencyList.push_back(oneNodeList);
52      }
53  }
54
55  void BruteForce::allPaths(int init_node, int fin_node, vector<int>path
        )
56  {
57      path.push_back(init_node);
58
59      if (init_node == fin_node)
60      {
61          // path found
62          if (path.size() == mAdjacencyList.size())
63          {
64              int l = 0, n = static_cast<int>(path.size()), i;
65              for (i = 0; i < n - 1; ++i)
66              {
67                  l += mAmatrix->at(path.at(i), path.at(i + 1));
68              }
69              l += mAmatrix->at(path.at(i), path.at(0));
70              if (l < lmin)
71              {
72                  lmin = l;
73                  minRoute = path;
74              }
75          }
76          return;
77      }
78
79      if (mAdjacencyList[init_node].empty())
80          return; // path not found
81
82      for (auto subnode:mAdjacencyList[init_node])
83      {
84          if (!isIn(path, subnode))
85              allPaths(subnode, fin_node, path);
86      }
87      return;
88  }
89
90  vector<int> BruteForce::getMinRouteVec()
91  {
92      return minRoute;
93  }
94
95  void BruteForce::printVec(vector<int> &vec)
```

```
96 {
97     for (unsigned int i = 0; i < vec.size(); ++i)
98     {
99         cout << vec.at(i) << "\t";
100    }
101    cout << endl;
102 }
103
104 bool BruteForce::isIn(vector<int>v, int elem)
105 {
106    for (unsigned int i = 0; i < v.size(); ++i)
107        if (v.at(i) == elem)
108            return true;
109    return false;
110 }
```

Тесты

Тестирование проводилось с помощью модуля QtTest. Для этого был написан класс TestSalesman. Сначала каждый алгоритм тестировался по отдельности на заранее заготовленном наборе тестовых данных . После этого на матрицах смежности размерности 6 x 6 проводилось сравнение результата, полученного муравьиным алгоритмом, и результата, полученного в результате полного перебора.

В тестировании для муравьиного алгоритма использовались следующие параметры.

1. $\alpha = 1$.
2. $\beta = 9$.
3. $\rho = 0.1$.
4. $t_{max} = 100$.
5. $\tau_{min} = 0.001$.
6. $\tau_{start} = \frac{1}{}$.
7. $Q = 350$.

Все написанные тесты были пройдены.

Использованный набор тестовых данных приведен в таблице 1 и в таблице 2.

Таблица 1: Набор тестовых данных

Матрица смежности	Ожидаемый результат
$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$	2
$\begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$	3
$\begin{bmatrix} 0 & 3 & 1 \\ 3 & 0 & 2 \\ 1 & 2 & 0 \end{bmatrix}$	6
$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$	4
$\begin{bmatrix} 0 & 2 & 1 & 1 \\ 2 & 0 & 1 & 1 \\ 1 & 1 & 0 & 2 \\ 1 & 1 & 2 & 0 \end{bmatrix}$	4
$\begin{bmatrix} 0 & 1 & 4 & 6 \\ 1 & 0 & 5 & 2 \\ 4 & 5 & 0 & 3 \\ 6 & 2 & 3 & 0 \end{bmatrix}$	10
$\begin{bmatrix} 0 & 4 & 4 & 1 \\ 4 & 0 & 1 & 2 \\ 4 & 1 & 0 & 4 \\ 1 & 2 & 4 & 0 \end{bmatrix}$	8
$\begin{bmatrix} 0 & 4 & 4 & 1 \\ 4 & 0 & 3 & 2 \\ 4 & 3 & 0 & 4 \\ 1 & 2 & 4 & 0 \end{bmatrix}$	10
$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}$	5
$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 5 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 5 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}$	5

Таблица 2: Набор тестовых данных (продолжение)

Матрица смежности	Ожидаемый результат
$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 5 & 1 \\ 1 & 1 & 0 & 5 & 1 \\ 1 & 5 & 5 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}$	5
$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 5 & 5 \\ 1 & 1 & 0 & 5 & 5 \\ 1 & 5 & 5 & 0 & 1 \\ 1 & 5 & 5 & 1 & 0 \end{bmatrix}$	9
$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 2 & 1 & 1 & 1 \\ 1 & 2 & 0 & 2 & 1 & 1 \\ 1 & 1 & 2 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$	6
$\begin{bmatrix} 0 & 2 & 1 & 1 & 1 & 1 \\ 2 & 0 & 2 & 1 & 1 & 1 \\ 1 & 2 & 0 & 2 & 1 & 1 \\ 1 & 1 & 2 & 0 & 2 & 1 \\ 1 & 1 & 1 & 2 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$	6

Экспериментальная часть

В данном разделе будет проведена параметризация муравьиного алгоритма. Были выбраны три матрицы смежности размером 10 x 10.

Первая матрица (M_1):

$$\begin{bmatrix} 0 & 40 & 30 & 10 & 70 & 25 & 50 & 60 & 40 & 35 \\ 40 & 0 & 70 & 90 & 40 & 50 & 80 & 50 & 45 & 40 \\ 30 & 70 & 0 & 60 & 20 & 80 & 40 & 60 & 65 & 80 \\ 10 & 90 & 60 & 0 & 50 & 10 & 80 & 15 & 90 & 30 \\ 70 & 40 & 20 & 50 & 0 & 75 & 30 & 30 & 40 & 25 \\ 25 & 50 & 80 & 10 & 75 & 0 & 10 & 20 & 10 & 70 \\ 50 & 80 & 40 & 80 & 30 & 10 & 0 & 70 & 40 & 30 \\ 60 & 50 & 60 & 15 & 30 & 20 & 70 & 0 & 20 & 80 \\ 40 & 45 & 65 & 90 & 40 & 10 & 40 & 20 & 0 & 90 \\ 35 & 40 & 80 & 30 & 25 & 70 & 30 & 80 & 90 & 0 \end{bmatrix}$$

Вторая матрица (M_2):

$$\begin{bmatrix} 0 & 30 & 40 & 10 & 70 & 25 & 50 & 60 & 40 & 20 \\ 30 & 0 & 70 & 90 & 40 & 50 & 80 & 50 & 45 & 40 \\ 40 & 70 & 0 & 30 & 50 & 80 & 40 & 60 & 65 & 80 \\ 10 & 90 & 30 & 0 & 50 & 10 & 80 & 15 & 90 & 30 \\ 70 & 40 & 50 & 50 & 0 & 75 & 50 & 30 & 40 & 25 \\ 25 & 50 & 80 & 10 & 75 & 0 & 5 & 20 & 10 & 70 \\ 50 & 80 & 40 & 80 & 50 & 5 & 0 & 70 & 40 & 30 \\ 60 & 50 & 60 & 15 & 30 & 20 & 70 & 0 & 20 & 80 \\ 40 & 45 & 65 & 90 & 40 & 10 & 40 & 20 & 0 & 75 \\ 20 & 40 & 80 & 30 & 25 & 70 & 30 & 80 & 75 & 0 \end{bmatrix}$$

Третья матрица (M_3):

$$\begin{bmatrix} 0 & 30 & 30 & 20 & 70 & 25 & 50 & 60 & 40 & 30 \\ 30 & 0 & 70 & 90 & 40 & 50 & 80 & 50 & 45 & 40 \\ 30 & 70 & 0 & 50 & 25 & 80 & 40 & 60 & 65 & 40 \\ 20 & 90 & 50 & 0 & 40 & 10 & 80 & 15 & 90 & 35 \\ 70 & 40 & 25 & 40 & 0 & 75 & 30 & 30 & 40 & 25 \\ 25 & 50 & 80 & 10 & 75 & 0 & 10 & 20 & 10 & 60 \\ 50 & 80 & 40 & 80 & 30 & 10 & 0 & 70 & 40 & 30 \\ 60 & 50 & 60 & 15 & 30 & 20 & 70 & 0 & 20 & 70 \\ 40 & 45 & 65 & 90 & 40 & 10 & 40 & 20 & 0 & 80 \\ 30 & 40 & 40 & 35 & 25 & 60 & 30 & 70 & 80 & 0 \end{bmatrix}$$

Постановка эксперимента

В рамках данной работы были проведены следующие эксперименты.

1. Для различных значений параметров α, β, ρ и t_{max} для кадой из трех матриц смежности M_1, M_2 и M_3 с помощью муравьиного алгоритма была найдена некоторая длина маршрута.

2. С помощью перебора был найен самый короткий маршрут для кадой из трех матриц смежности M_1, M_2 и M_3.

3. Было проведено сравнение длины маршута, полученной с помощью муравьиного алгоритма и с помощью полного перебора.

4. На основании сравнения были выбраны наилучшие сочетания параметров муравьиного алгоритма на данном классе данных.

Значения параметров изменялись в следующих диапазонах.

1. $\alpha \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

2. $\beta + \alpha = \text{const}$.

3. $\rho \in \{0.1, 0.2, 0.25, 0.3, 0.4, 0.5, 0.6, 0.7, 0.75, 0.8, 0.9\}$.

4. $t_{max} \in \{5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$.

Измерения проводились на компьютере HP Pavilion Notebook на базе Intel Core i5-7200U, 2.50 Гц с 6 Гб оперативной памяти и с 4 логическими ядрами под управлением операционной системы Linux Mint.

Параметризация муравьиного алгоритма на основании проведенного эксперимента

В таблице 3 приведены сочетания параметров, при которых муравьный алгоритм давал наиболее оптимальное решение для всех трех матриц. Условные обозначения в таблице 3:

1. B1 - длина маршрута для матрицы M_1, рассчитанная с помощью полгого перебора;

2. A1 - длина маршрута для матрицы M_1, рассчитанная с помощью муравьиного алгоритма;

3. B2 - длина маршрута для матрицы M_2, рассчитанная с помощью полгого перебора;

4. A2 - длина маршрута для матрицы M_2, рассчитанная с помощью муравьиного алгоритма;;

5. B3 - длина маршрута для матрицы M_3, рассчитанная с помощью полгого перебора;

6. A3 - длина маршрута для матрицы M_4, рассчитанная с помощью муравьиного алгоритма;.

Таблица 3: Наиболее оптимальные сочетания параметров

α	β	ρ	t_{max}	B1	A1	B2	A2	B3	A3
4	6	0.6	20	225	225	240	240	235	240
6	4	0.3	40	225	225	240	245	235	240
1	9	0.7	50	225	225	240	245	235	240
6	4	0.9	70	225	230	240	245	235	240
3	7	0.6	80	225	230	240	245	235	240
6	4	0.25	90	225	230	240	240	235	235

Выводы

Таким образом, для заданного класса данных были найдены параметры, которые обеспечивают наиболее оптимальное решение.

Заключение

Таким образом, в ходе выполнения данной лабораторной работы был приобретен навык параметризации метода на примере решения задачи коммивояжёра методом, основанным на муравьином алгоритме: был произведен отбор таких сочетаний параметров, при котором муравьиный алгоритм решает заданный класс задач наиболее точно.

Список литературы

1. Ульянов М. В. Ресурсно-эффективные компьютерные алгоритмы. Разработка и анализ. -М.:Наука, 2007.