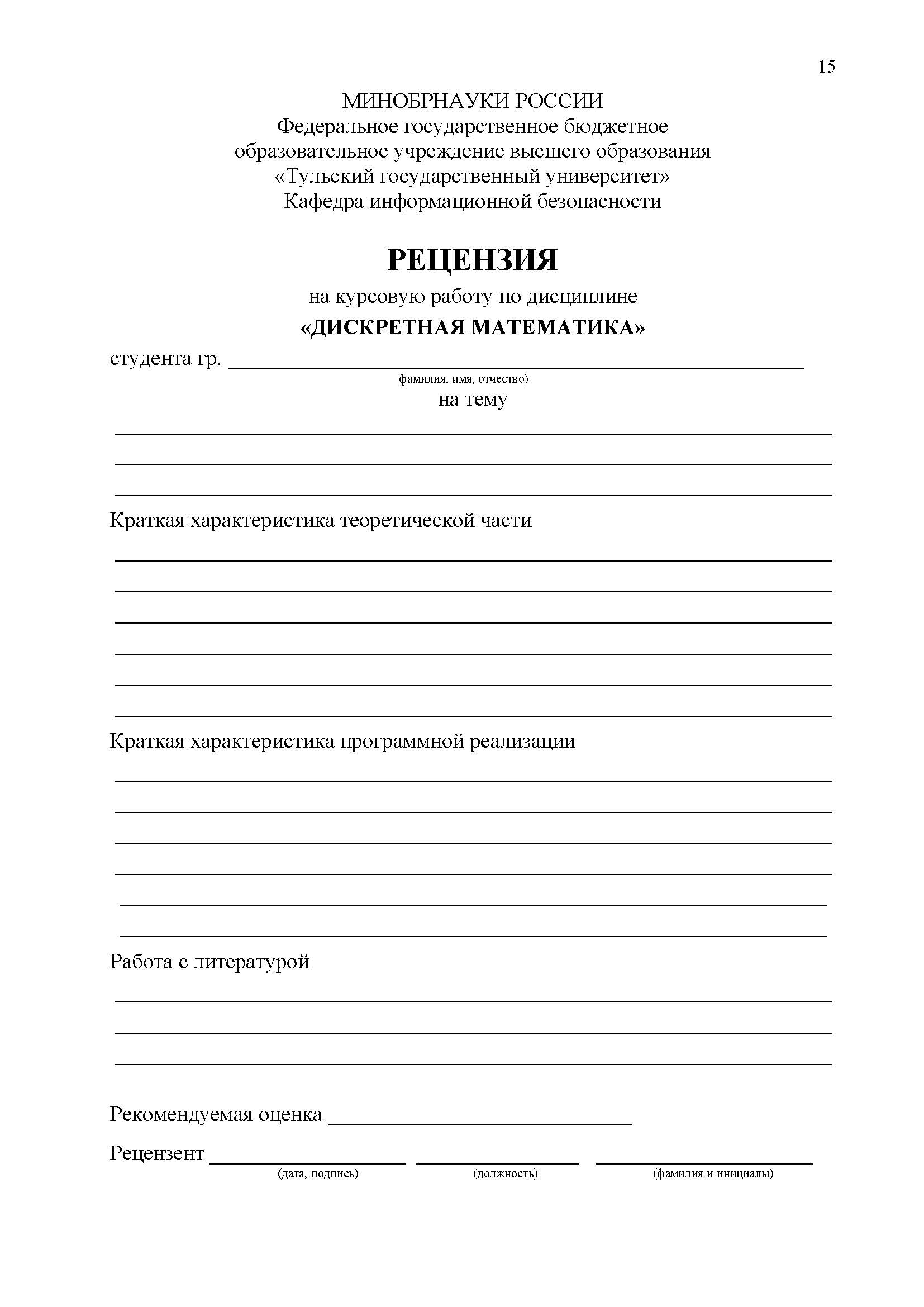
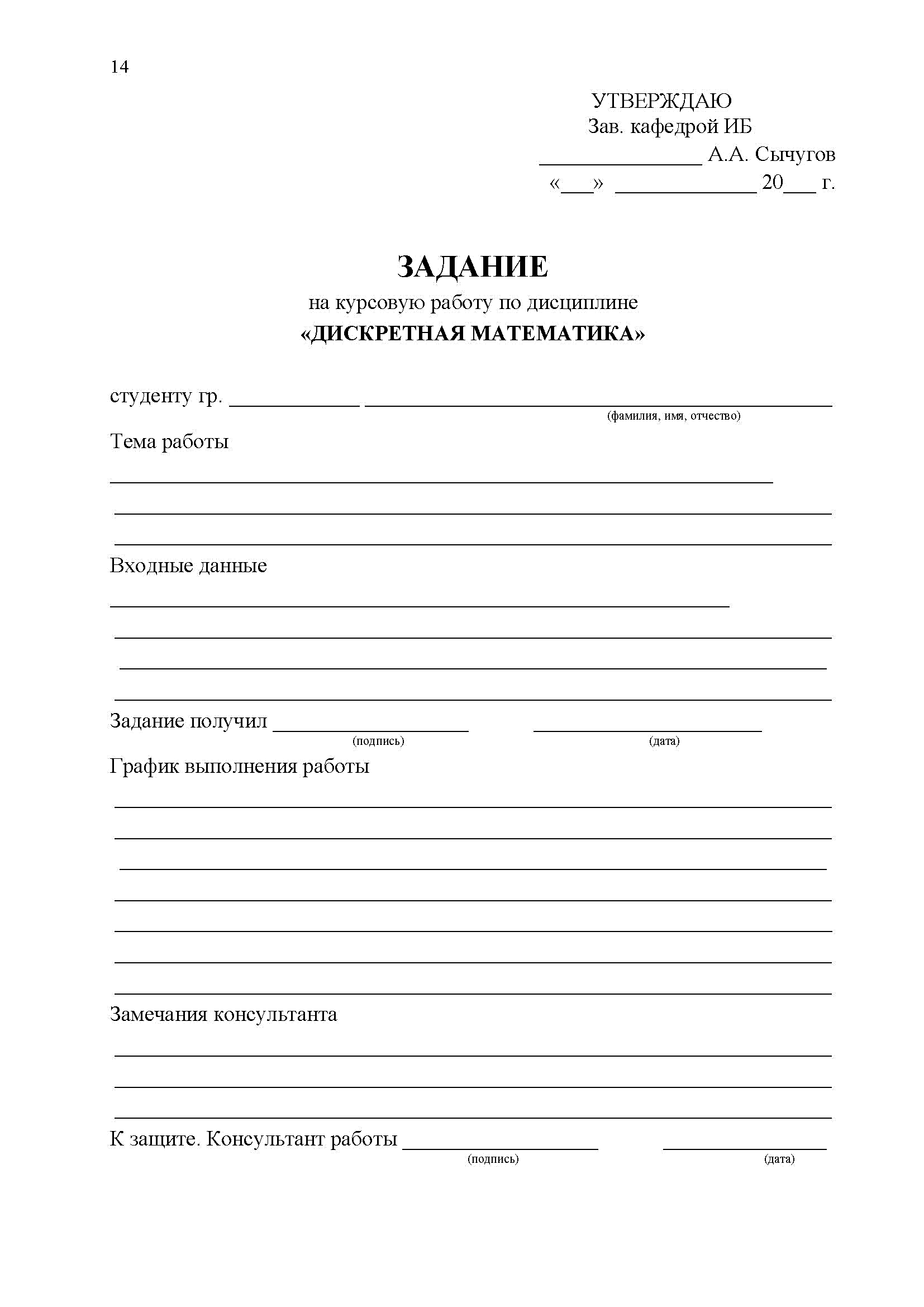


«Гамильтоновы циклы и задача коммивояжёра»

22

Павлова В.С.

230711



«Гамильтоновы циклы и задача коммивояжёра»

Павловой Виктории Сергеевне

230711

# **ЗАДАНИЕ**

# **ОГЛАВЛЕНИЕ**

[**ЗАДАНИЕ** 2](file:///C:\Users\Вика\Desktop\КУРСОВАЯ%20ДМ\КР.docx#_Toc119436202)

[**ОГЛАВЛЕНИЕ** 4](#_Toc119436203)

[**ВВЕДЕНИЕ** 5](#_Toc119436204)

[**ГЛАВА 1. ПОСТАНОВКА ЗАДАЧИ** 6](#_Toc119436205)

[**ГЛАВА 2. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ** 7](#_Toc119436206)

[2.1. Основные понятия теории графов 7](#_Toc119436207)

[2.2. Гамильтоновы графы, циклы и пути 9](#_Toc119436208)

[2.3. Постановка задачи коммивояжёра 12](#_Toc119436209)

[**ГЛАВА 3. ПРАКТИЧЕСКАЯ ЧАСТЬ** 14](#_Toc119436210)

[3.1. Алгоритмы решения задачи коммивояжёра 14](#_Toc119436211)

[3.1.1. Алгоритм ближайшего соседа 14](#_Toc119436212)

[3.1.2. Метод ветвей и границ 18](#_Toc119436213)

[3.1.3. Муравьиный алгоритм 24](#_Toc119436214)

[3.1.4. Сравнительная характеристика приведённых методов 29](#_Toc119436215)

[**ЗАКЛЮЧЕНИЕ** 32](#_Toc119436216)

[**БИБЛИОГРАФИЧЕСКИЙ СПИСОК** 33](#_Toc119436217)

[**ПРИЛОЖЕНИЕ 1** 34](#_Toc119436218)

# **ВВЕДЕНИЕ**

Мир современного человека представляет собой запутанную сеть сложных и взаимосвязанных структур. В нём нет ничего, что могло бы существовать в этом мире исключительно само по себе. Понимание взаимосвязей между объектами и явлениями способно дать ответы на многие вопросы, возникающие в самых разных научных дисциплинах. Одним из инструментов анализа процессов является абстракция, а именно представление с помощью графов. Это обосновывает актуальность выбранной темы: теория графов позволяет отражать любые системы, где есть те или иные взаимосвязи, а также даёт возможность их анализировать и оценивать. Так, например, именно графами образованы генеалогические деревья, иерархические структуры и даже карты движения общественного транспорта и населённых пунктов.

Теория графов в настоящее время является интенсивно развивающимся разделом математики, она применяется в машинном обучении и является востребованным инструментом при создании искусственного интеллекта. Вопрос оптимизации различных процессов стоит особенно остро для постоянно развивающегося информационного общества. Одной из актуальных задач этой области является так называемая задача коммивояжёра, или задача посыльного, заключающаяся в необходимости найти кратчайший путь между конечным множеством мест, расстояние между которыми известно. В качестве объекта исследования для данной курсовой работы был выбран гамильтонов цикл, тесно связанный с этой задачей, а в качестве предмета исследования – алгоритм нахождения гамильтонова цикла на примере задачи о коммивояжёре.

Целью курсовой работы является разработка алгоритма нахождения гамильтонового цикла минимального веса, где под весом цикла понимается сумма весов составляющих его ребер. Для достижения данной цели были поставлены конкретные задачи, определённые в разделе «1 Постановка задачи».

# **ГЛАВА 1. ПОСТАНОВКА ЗАДАЧИ**

Для изучения гамильтоновых циклов в графах были поставлены следующие задачи:

1. Определить сущность гамильтонового цикла. Для этого необходимо:

а) дать определение понятиям «граф», «гамильтонов граф», «путь в графе», «гамильтонов цикл»;

б) рассмотреть алгоритм нахождения гамильтонового цикла в графе и его реализацию;

1. Реализовать в программе алгоритм нахождения приемлемого решения задачи коммивояжёра. Для этого необходимо:

а) рассмотреть алгоритм ближайшего соседа;   
б) рассмотреть метод ветвей и границ;

в) рассмотреть муравьиный алгоритм.

1. Сделать выводы об итогах проведённого исследования и эффективности использованных алгоритмов.

# **ГЛАВА 2. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ**

## 2.1. Основные понятия теории графов

В качестве фундаментального понятия необходимо дать теоретико-множественное определение графа. В теории графов математический объект, который представляет собой совокупность двух множеств — множества объектов, называемого множеством вершин, и множества их парных связей, называемого множеством рёбер, — называется графом. Таким образом, *граф* — это два конечных непересекающихся множества, причём второе состоит из двухэлементных подмножеств первого. Элемент множества рёбер есть пара элементов множества вершин. Принято следующее обозначение: граф G = (V, E), где пара (V, E) называется неориентированным графом, если V – непустое множество вершин, а E – множество неупорядоченных пар различных рёбер.

Существуют различные способы задания графов. Первый из них основан на использовании множеств, также применяют матричный способ и графический. Пусть дан граф G = (X, U) с множеством вершин X = {} и множеством рёбер U = {{},{},{},{},{},{}, {}}. [1]

Имея теоретико-множественное определение неориентированного графа, можно изобразить его с помощью рисунка (диаграммы), содержащего вершины и рёбра. Графическое изображение графа G = (V, E) представлено на рисунке 1.

  
Рисунок 2.1 – Графическое представление графа G

Матричный способ задания неориентированного графа подразумевает составление матрицы смежности S = [], которая определяется следующим образом:

, если существует ребро ,

, если ребра нет.

Матрица смежности S графа G представлена в таблице 2.1. Как видно из таблицы, матрица смежности для неориентированного графа обладает симметрией относительно главной диагонали. Если дан орграф, то есть его рёбра являются дугами и имеют направление, то принято рассматривать матрицу инциденций C = [], которая определяется следующим образом:

, если является началом дуги ,

, если является концом дуги ,

, если не является ни концом, ни началом дуги , или если является петлёй.

Матрица инциденций существует и для неориентированного графа (таблица 2.2), хотя и не является столь информативной, как в случае орграфа. [2]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | Таблица 2.1 – Матрица смежности S графа G. | | | | | | |  |  |  |  |  |  | |  | 0 | 1 | 1 | 1 | 0 | |  | 1 | 0 | 1 | 0 | 1 | |  | 1 | 1 | 0 | 1 | 0 | |  | 1 | 0 | 1 | 0 | 1 | |  | 0 | 1 | 0 | 1 | 0 | | |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | | Таблица 2.2 – Матрица инциденций C графа G. | | | | | | | | | |  |  |  |  |  |  |  |  | |  | 0 | 0 | 0 | 0 | 1 | 1 | 1 | |  | 1 | 1 | 0 | 0 | 0 | 1 | 0 | |  | 1 | 0 | 0 | 1 | 0 | 0 | 0 | |  | 0 | 0 | 1 | 1 | 1 | 0 | 0 | |  | 0 | 1 | 1 | 0 | 0 | 0 | 1 | |

## 2.2. Гамильтоновы графы, циклы и пути

Если граф имеет простой (не содержащий петель) цикл, содержащий все вершины графа по одному разу, то такой цикл называется *гамильтоновым циклом*, а граф – *гамильтоновым графом*. Путем или цепью в графе называют конечную последовательность вершин, в которой каждая вершина (кроме последней) соединена со следующей в последовательности вершин ребром.

Граф, который содержит простой путь, проходящий через каждую его вершину, называется полугамильтоновым. Это определение можно распространить на ориентированные графы, если путь считать ориентированным. [3]

|  |  |  |
| --- | --- | --- |
| а) | б) | в) |

Рисунок 2.2 – Гамильтонов граф (а), полугамильтонов граф (б) и негамильтонов граф (в)

Гамильтоновы путь, цикл и граф названы в честь ирландского математика Уильяма Гамильтона, который занимался исследованием задачи «кругосветного путешествия» по додекаэдру (рисунок 3, а). В его задаче вершины додекаэдра символизировали города, а рёбра — соединяющие их дороги. Путешествующий должен пройти «вокруг света», найдя путь, который проходит через все вершины ровно один раз. В качестве модели для решения данной задачи им был предложен граф с двадцатью вершинами (рисунок 2.3, б). [4]

|  |  |
| --- | --- |
| а) | б) |

Рисунок 2.3 – Додекаэдр (а) и плоский граф, ему изоморфный (б)

Важным замечанием является тот факт, что не в каждом графе существует гамильтонов цикл. Впервые условие, из которого следовало бы существование такого цикла, сформулировал английский математик и физик Поль Дирак, и звучит оно следующим образом: если каждая вершина графа соединена рёбрами более чем с половиной других, то в нём есть гамильтонов цикл.

Более строго необходимое и достаточное условие существование гамильтонова цикла в графе можно сформулировать так: если неориентированный граф G содержит гамильтонов цикл, тогда в нём не существует ни одной вершины с локальной степенью Под локальной степенью вершины понимается число рёбер, ей инцидентных. [5]

За основу алгоритма для нахождения гамильтонового цикла в невзвешенном графе возьмём так называемый перебор с возвратом. Такой подход позволяет к уже выстроенному отрезку маршрута прибавляться лишь те вершины, которые соединяются ребром с конечной вершиной маршрута и не посещались раньше. После прибавления новой вершины к пути происходит рекурсия, то есть запуск алгоритма из новой вершины. В случае возврата пути из какой-либо вершины на предыдущую точку, с оставленной вершины удаляется пометка о её посещении. При таком способе возможен возврат алгоритма на эту же вершину повторно, но уже другим путём. Более того, это обязательно случится, если гамильтонов цикл существует в рассматриваемом графе, поскольку путь должен пройти через каждую вершину.

Реализация данного алгоритма на языке программирования C++ представлена в листинге 1. В программе используется пользовательский класс Graph и методы работы с ним. Реализация данного класса приведена в листинге 1 приложения 1.

**Листинг 2.1 – Программа для поиска гамильтонова цикла в графе**

#include <iostream>

#include "Graph.h"

#include <Windows.h>

#include <vector>

using namespace std;

vector <bool> visited; //массив для помечания пройденных вершин

vector <int> path; //массив, хранящий гамильтонов путь

//в функцию передаётся начальная вершина cur и сам граф graph

bool FindGamilton(Graph graph, int cur)

{

//считывание матрицы смежности

vector <vector <int>> matrix = graph.MakeMatrixFromList();

//добавление начальной вершины в путь

path.push\_back(cur);

if (path.size() == graph.nodeCount) //если пройдены все вершины

{

if (matrix[path[0]][path.back()] == 1) //если начало и конец связаны

{

path.push\_back(path[0]); //замкнуть цикл

return true; //гамильтонов цикл найден, вернуть true

}

else

{

path.pop\_back(); //удалить вершину из пути

return false; //гамильтонов цикл не существует

}

}

visited[cur] = true; //если все вершины ещё не пройдены, двигаться далее

for (int i = 0; i < graph.nodeCount; i++) //проход по матрице смежности

{

//есть ребро между текущей вершиной и следующей

// при этом следующая не посещена

if (matrix[cur][i] == 1 && !visited[i])

if (FindGamilton(graph, i)) //запускаем поиск из следующей вершины

return true;

}

visited[cur] = false; //выход из рекурсии, если цикл не найден

path.pop\_back();

return false;

}

**Листинг 2.1 (продолжение)**

int main()

{

setlocale(LC\_ALL, "RUSSIAN");

Graph graph;

visited.resize(graph.nodeCount, false); //отметить все вершины непройденными

path.clear(); //очистить путь

graph.ReadMatrix(); //считать матрицу смежности

if (FindGamilton(graph, 0)) //если гамильтонов цикл существует

{

cout << "\n\n\t\t\t\tГамильтонов цикл найден: ";

int i = 0;

for (auto to : path) //вывод цикла

{

cout << to + 1;

if (i < path.size() - 1) cout << " -> ";

i++;

} cout << "\n\n";

}

return 0;

}

## 2.3. Постановка задачи коммивояжёра

В контексте изучения гамильтоновых циклов рассмотрим хорошо известную транспортную задачу коммивояжера, которая состоит в следующем: коммивояжер должен посетить каждый из заданных городов по одному разу, выехав из некоторого из этих городов и вернувшись в него же. Требуется найти кратчайший маршрут, зная расстояния между каждой парой городов. Математическая постановка этой задачи такова: в полном взвешенном графе требуется найти гамильтонов цикл минимального веса. Под весом цикла понимается сумма весов составляющих его ребер. [6]

Для решения этой задачи существует и простой алгоритм – полный перебор всех возможных вариантов. Однако, такой подход, как правило, неприемлем из-за сложности порядка . Построение гамильтонова цикла — непростая задача, и в настоящее время неизвестно алгоритма его решения со сложностью O(*n*). Более того, такого алгоритма, вероятно, пока вообще не существует – это одна из нерешённых проблем теории сложности алгоритмов. [6]

В зависимости от критерия выгодности маршрута выделяют разные виды задачи коммивояжёра, важнейшими из которых являются симметричная и метрическая задачи. В случае *симметричной* задачи все пары ребер между одними и теми же вершинами имеют одинаковую длину, то есть для ребра ( одинаковы расстояния . Иначе говоря, задача моделируется неориентированным графом. Симметричную задачу коммивояжёра называют *метрической*, если относительно длин ребер выполняется неравенство треугольника, то есть обходные пути длиннее прямых, и ребро ( никогда не бывает длиннее через промежуточную вершину . [7]

Более приближенной к реальности является так называемая асимметричная задача коммивояжёра, в которой необходимо учитывать также направление рёбер, поскольку она моделируется орграфом. К тому же в случае реальных городов и дорог большое значение имеют побочные факторы, такие как состояние дорожных покрытий, скопление автомобилей, аварии, направление движения и так далее.

Для определённости уточним, что если не оговорено иное, по умолчанию будем рассматривать только полный взвешенный граф, то есть простой (не имеющий кратных рёбер и петель) неориентированный граф, в котором каждая пара различных вершин смежна. Тогда рассматриваемая задача коммивояжёра сводится к виду симметричной, поскольку все пары ребер между одними и теми же вершинами будут иметь одинаковую длину.

# **III. ПРАКТИЧЕСКАЯ ЧАСТЬ**

## Алгоритмы решения задачи коммивояжёра

### ***Алгоритм ближайшего соседа***

Как было сказано ранее в главе 2, эффективный алгоритм решения задачи коммивояжёра пока не известен. Для сложных графов количество гамильтоновых циклов, которые необходимо рассмотреть для нахождения самого оптимального, невероятно велико и быстро растёт в зависимости от количества вершин. Так, у алгоритма полного перебора, также называемого алгоритмом грубой силы (англ. Brut force) сложность равна . На практике чаще применяются алгоритмы поиска *приемлемого* решения. Такое решение называется *субоптимальным* и не всегда даёт цикл минимального общего веса, однако результат его оказывается вполне удовлетворителен и хорош в сравнении с другими путями. Один из наиболее простых подобных алгоритмов – жадный алгоритм (англ. Greedy algorithm) или **алгоритм ближайшего соседа**. [8]

Идея данного алгоритма основана на следующем условии: если на каждом шаге идти в ближайшую вершину, то конечный маршрут получится довольно хорош в целом. Перед коммивояжёром ставится задача посещать ближайший (то есть прибавляющий минимальный вес) из ещё не пройденных пунктов. Одним из эвристических критериев оценки решения является правило: если путь, пройденный на последних шагах алгоритма, сравним с путём, пройденным на начальных шагах, то можно условно считать найденный маршрут приемлемым, иначе, вероятно, существуют более оптимальные решения. [9]

Смысл данного алгоритма выражена блок-схемой, представленной на рисунке 3.1.



Рисунок 3.1 – блок-схема алгоритма ближайшего соседа

Алгоритм ближайшего соседа подразумевает под собой принятие локально оптимальных решений по выбору следующей вершины с допущением, что конечное решение тоже окажется оптимальным. Отмечая пройденные вершины, мы формируем маршрут, параллельно прибавляя к нему вес пройденного пути с каждой новой вершиной. [10] Рассмотрим граф G, приведённый на рисунке 3.2.

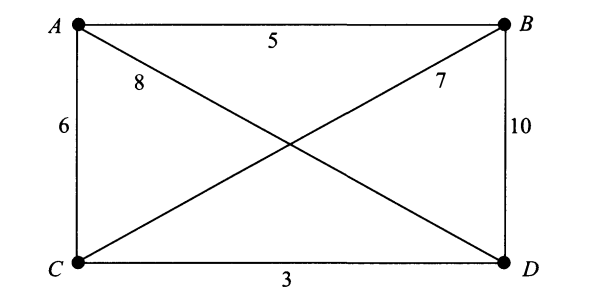


Рисунок 3.2 – взвешенный связный граф G

Если перебрать все возможные варианты путей в графе, становится ясно, что для него существует несколько гамильтоновы циклов разного веса:

1. Цикл DCABD веса 24;
2. Цикл ABCDA веса 23 (наиболее оптимальный маршрут);
3. Цикл ACBDA веса 31.

Отсюда вытекает основная проблема алгоритма ближайшего соседа – необходимость запускать цикл из каждой вершины, чтобы найти лучший вариант маршрута. Для любого количества городов, большего трёх, в задаче коммивояжёра можно подобрать такое расположение городов (значение расстояний между вершинами графа и указание начальной вершины), что алгоритм ближайшего соседа будет выдавать наихудшее решение. [9]

Программная реализация этого алгоритма несколько схожа с рассмотренной ранее для поиска гамильтонова цикла (Гл. 2.1) в невзвешенном связном графе. Для решения задачи коммивояжёра необходимо внести в неё некоторые коррективы, чтобы учитывался вес ребра. Программная реализация метода ближайшего соседа для решения задачи коммивояжёра на языке программирования C++ представлена в листинге 3.1.

**Листинг 3.1 – Программа для решения задачи коммивояжёра методом ближайшего соседа**

#include <iostream>

#include "Graph.h"

#include <Windows.h>

#include <vector>

#include <fstream>

using namespace std;

int weight = 0; //поле для хранения веса маршрута

vector <bool> visited; //массив для помечания пройденных вершин

vector <int> path; //массив, хранящий найденный путь

int FindGamiltonWithWeight(Graph graph, int cur)

{

vector <vector <int>> matrix = graph.MakeMatrixFromList(); //матрица смежности

path.push\_back(cur); //пометили начальную вершину

if (path.size() == graph.nodeCount) //если пройдены все вершины

{

if (matrix[path[0]][path.back()] != 0) //если цикл замкнут

{

weight += matrix[path[0]][path.back()]; //внести вес последнего ребра

path.push\_back(path[0]); //пометить последнюю вершину

return weight; //вернуть вес найденного пути

}

lse { path.pop\_back(); //отметить вершину непройденной

**Листинг 3.1 – Программа для решения задачи коммивояжёра методом ближайшего соседа (продолжение)**

return -1;

}

}

visited[cur] = true; //отметить вершину пройденной

for (int i = 0; i < graph.nodeCount; i++) //проход по матрице смежности

{

//если есть ребро и следующая вершина не была пройдена

if (matrix[cur][i] != 0 && !visited[i])

{

//внести вес новой пройденный вершины в общий вес пути

weight += matrix[cur][i];

//запуск алгоритма из новой вершины

if (FindGamiltonWithWeight(graph, i) != -1)

return weight;

}

}

visited[cur] = false; //отметить вершину непройденной

path.pop\_back(); //исключить её из пути

return -1;

}

int main()

{

setlocale(LC\_ALL, "RUSSIAN");

Graph graph;

graph.ReadMatrix();

int minWeight = INT\_MAX; //поле для хранения мин. веса пути

vector <int> bestPath; //массив для записи наилучшего пути

int k = 0;

while (k < graph.nodeCount) //пока не перебраны все варианты

{

visited.clear(); //отметить все вершины не пройденными

visited.resize(graph.nodeCount, false);

path.clear(); //очистить путь

weight = 0;

//если существует гамильтонов цикл

if (FindGamiltonWithWeight(graph, k) != -1)

{

if (weight < minWeight) //если вес является минимальным

{

minWeight = weight; //зафиксировать найденный вес пути

bestPath = path; //зафиксировать найденный путь

}

}

k++;

}

cout << "\n\n\t\t\t\tГамильтонов цикл найден. Его вес: " << minWeight << "\n";

int i = 0;

cout << "\n\n\t\t\t\tМаршрут: ";

for (auto to : bestPath)

{

cout << to + 1;

if (i < bestPath.size() - 1) cout << " -> ";

i++;

} cout << "\n\n";   
return 0;}

### ***Метод ветвей и границ***

В процессе перебора вариантов всегда желательно рассматривать не все из них, а лишь те, которые следует считать перспективными и отбрасывать бесперспективные. Возникает вопрос, как же осуществлять такой отбор. Одним из подходов к поиску оптимальных решений является метод ветвей и границ (англ. Branch and bound), который позволяет найти точное решение задачи коммивояжера, вычислив длины всех возможных маршрутов и выбирав маршрут с наименьшей длиной. [7]

Суть данного метода сводится к следующему: среди всех допустимых на текущем этапе решений выбирается оптимальное, где критерием выбора является длина гамильтонова цикла. Схематически применение метода ветвей и границ можно представить в виде дерева (рис. 3.3), корнем которого является множество всех решений, а листьями – гамильтоновы циклы.



Рисунок 3.3 – Дерево решений метода ветвей и границ

Использование понятия границ и их расчёт стимулирует или тормозит рост ветвей в таком дереве. [7] Внутренние узлы дерева представляют множества решений, объединённые какой-то общей характеристикой. В качестве общей характеристики выступает, как правило, либо часть потенциального решения (например, уже построенная часть гамильтонова цикла), либо ограничение на включение или невключение элементов (списки рёбер, входящих и не входящих в цикл из данного множества). Такое множество потенциальных решений называется *ветвью*. [11]

Находясь во внутренних вершинах дерева перебора, алгоритм вычисляет оценки (*нижние границы*) стоимости решений, входящих в соответствующую ветвь. Для задач минимизации вычисляется оценка снизу, для максимизации — сверху. Если вычисленная оценка хуже, чем стоимость уже найденного рекордного значения, то алгоритм отсекает текущую ветвь, то есть исключает из перебора все входящие в неё потенциальные решения. Операция отсечения является ключевой для метода ветвей и границ, поскольку позволяет, при правильном выборе способа ветвления и вычисления границ значительно сокращать количество обрабатываемых вариантов. [11]

Рассмотрим данный алгоритм на примере взвешенного графа G, заданного своей матрицей смежности S. На главной диагонали матрицы помещены прочерки, поскольку диагональные клетки не рассматриваются (петель нет).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Таблица 3.1 – Матрица смежности S графа G | | | | | |
|  |  |  |  |  |  |
|  | - | 20 | 18 | 12 | 8 |
|  | 5 | - | 14 | 7 | 11 |
|  | 12 | 18 | - | 6 | 11 |
|  | 11 | 17 | 11 | - | 12 |
|  | 5 | 5 | 5 | 5 | - |

Вычислим нижнюю границу для корневой вершины как сумму констант приведения для столбцов и строк (данная граница будет являться стоимостью, меньше которой невозможно построить искомый маршрут). [12]

1. Найдём минимум для каждой из *строк* и произведём редуцирование каждого элемента строки матрицы следующим образом: если не лежит на диагонали, то
2. Найдём минимум для каждого из *столбцов* и произведём редуцирование для каждого элемента столбца матрицы следующим образом: если не лежит на диагонали, то
3. Вычислим нижнюю границу в корневой точке решения, как сумму найденных ранее минимумов.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  | **min** |
|  | - | 12 | 10 | 4 | 0 | **8** |
|  | 0 | - | 9 | 2 | 6 | **5** |
|  | 6 | 12 | - | 0 | 5 | **6** |
|  | 0 | 6 | 0 | - | 1 | **11** |
|  | 0 | 0 | 0 | 0 | - | **5** |
| **min** | **0** | **0** | **0** | **0** | **0** |  |

Рисунок 3.4 – Преобразование матрицы смежности

Таким образом, нижняя граница корневого решения будет равна следующей сумме 8 + 5 + 6 + 11 + 5 + 0 = **35**. Проделывая аналогичные действия для преобразованной матрицы, получим следующее дерево решений (рисунок 3.5):



Рисунок 3.5 – Дерево решений для графа G

Длина итогового маршрута есть самое последнее значение локальной нижней границы правильной ветви решения, то есть L = 41. В приведённом примере в ходе решения были выбраны следующие рёбра: (4;2), (1;5), (2;1), (3;4), (5;3). Упорядочив и соединив эти отрезки, получим следующий маршрут: 1 -> 5 -> 3 -> 4 -> 2 -> 1.

Программная реализация данного алгоритма представлена в листинге 3.2.

**Листинг 3.2 – Программа для решения задачи коммивояжёра методом ветвей и границ**

#include <iostream>

#include "Graph.h"

using namespace std;

int minWeight = INT\_MAX;

int lowerBound = 0;

vector <int> bestPath;

vector <int> currentPath;

vector <bool> visited;

void FinalPath(vector <int> currentPath) //запись конечного маршрута

{

bestPath.clear();

for (int i = 0; i < currentPath.size(); i++)

bestPath.push\_back(currentPath[i]);

**Листинг 3.2 – Программа для решения задачи коммивояжёра методом ветвей и границ (продолжение)**

bestPath.push\_back(currentPath[0]);

return;

}

int FirstMin(vector <vector <int>> matrix, int k) //поиск первого подходящего ребра,

{ //ведущего в вершину k

int min = INT\_MAX;

for (int i = 0; i < matrix.size(); i++)

if (matrix[k][i] < min && k != i)

min = matrix[k][i];

return min;

}

int SecondMin(vector <vector <int>> matrix, int k) //поиск второго подходящего ребра

{ //ведущего в вершину k

int first = INT\_MAX, second = INT\_MAX;

for (int i = 0; i < matrix.size(); i++)

{

if (k!= i && matrix[k][i] <= first)

{

second = first;

first = matrix[k][i];

}

if (k != i && matrix[k][i] <= second

&& matrix[k][i] != first)

second = matrix[k][i];

}

return second;

}

int Solver(vector <vector <int>> matrix, //функция для поиска пути

int currentBound, int currentWeight,   
 int level, vector <int> currentPath)

{

if (level == matrix.size()) //текущий уровень дерева решений

{ //достиг ранга матрицы смежности

if (matrix[currentPath[level - 1]][currentPath[0]] != 0)

{

int currentRes = currentWeight +

matrix[currentPath[level - 1]][currentPath[0]];

if (currentRes < minWeight)

{

FinalPath(currentPath);

minWeight = currentRes;

}

}

return minWeight;

}

for (int i = 0; i < matrix.size(); i++) //ветвление не закончено

{

if (matrix[currentPath[level - 1]][i] != 0 //вершина не посещена

&& visited[i] == false)

{

int temp = currentBound; //запомнить текущую границу

currentWeight += matrix[currentPath[level - 1]][i];

//вычисление текущего веса пути

if (level == 1)

currentBound -= ((FirstMin(matrix, currentPath[level - 1]) +

**Листинг 3.2 – Программа для решения задачи коммивояжёра методом ветвей и границ (продолжение)**

FirstMin(matrix, i)) / 2);

else

currentBound -= ((SecondMin(matrix, currentPath[level - 1]) +

FirstMin(matrix, i)) / 2);

if (currentBound + currentWeight  
< minWeight) //если текущее ветвление выгодно

{ //продолжить путь из него

currentPath[level] = i;

visited[i] = true;

Solver(matrix, currentBound, currentWeight, level + 1,

currentPath);

}

currentWeight -= matrix[currentPath[level - 1]][i];

currentBound = temp;

visited.clear();

visited.resize(matrix.size(), false); //вернуться назад

for (int j = 0; j <= level - 1; j++)

visited[currentPath[j]] = true;

}

}

}

void BranchAndBoundSolver(vector <vector <int>> matrix)

{

minWeight = INT\_MAX; //сброс минимального веса пути

lowerBound = 0; //обнуление нижней границы

currentPath.resize(matrix[0].size(), -1); //массив для хранения текущего пути

visited.resize(matrix[0].size(), false); //массив для отметки пройденных вершин

int min;

for (int i = 0; i < matrix.size(); i++) //вычисление нижней границы

{

min = INT\_MAX;

for (int j = 0; j < matrix[i].size(); j++)

{

if (matrix[i][j] < min && matrix[i][j] != 0) min = matrix[i][j];

}

lowerBound += min;

}

visited[0] = true;

currentPath[0] = 0;

Solver(matrix, lowerBound, 0, 1, currentPath); //рекурсивный поиск пути

return;

}

int main()

{

setlocale(LC\_ALL, "RUSSIAN");

Graph graph;

graph.ReadMatrix();

vector <vector <int>> matrix = graph.MakeMatrixFromList();

BranchAndBoundSolver(matrix);

printf("Минимальная длина пути: %d\n", minWeight);

printf("Путь: ");

for (int i = 0; i < graph.nodeCount; i++)

printf("%d ", bestPath[i]);   
return 0;  
}

### ***Муравьиный алгоритм***

Основу так называемых муравьиных алгоритмов оптимизации составляет подражание самоорганизации муравьиной колонии. Колония муравьев может рассматриваться как многоагентная 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 представляют собой вероятностную жадную эвристику, где вероятности устанавливаются, исходя из информации о качестве решения, полученной из предыдущих решений. Они могут использоваться как для статических, так и для динамических комбинаторных оптимизационных задач. Сходимость гарантирована, то есть в любом случае мы получим оптимальное решение, хотя скорость сходимости неизвестна. [13]

Идея муравьиного алгоритма – моделирование поведения муравьёв, связанного с их способностью быстро находить кратчайший путь от муравейника к источнику пищи и адаптироваться к изменяющимся условиям, находя новый кратчайший путь. При своём движении муравей метит путь феромоном, и эта информация используется другими муравьями для выбора пути. Это элементарное правило поведения и определяет способность муравьёв находить новый путь, если старый оказывается недоступным. [13]

Моделирование поведения муравьёв связано с распределением феромона на тропе – ребре графа в задаче коммивояжёра. 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 рёбрах, следовательно, большее количество муравьёв будет включать его в синтез собственных маршрутов. Моделирование такого подхода, использующего только положительную обратную связь, приводит к преждевременной сходимости – большинство муравьёв двигается по локально оптимальному маршруту. Избежать этого можно, моделируя отрицательную обратную связь в виде испарения феромона. При этом если феромон испаряется быстро, то это приводит к потере памяти колонии и забыванию хороших решений. С другой стороны, большое время испарения может привести к получению устойчивого локального оптимального решения.

Таким образом, при решении задачи коммивояжёра муравьиным алгоритмом руководствуются следующими принципами:

1. Муравьи имеют собственную «память». Поскольку каждый город может быть посещён только один раз, то у каждого муравья есть список уже посещённых городов – список запретов. Тогда пусть – список городов, которые необходимо посетить муравью , находящемуся в городе .

2. Муравьи обладают «зрением» – у них есть эвристическое «желание» посетить город , если они находятся в городе . Принято считать, что видимость обратно пропорциональна расстоянию между городами ;

3. Муравьи обладают «обонянием» – они могут улавливать след феромона, подтверждающий желание посетить город из города на основании опыта других муравьёв. Количество феромона на ребре в момент времени обозначается как .

4. На основании вышеуказанных принципов сформулировано вероятностно-пропорциональное правило, определяющее вероятность перехода -ого муравья из города в город :

Где – параметры, задающие веса следа феромона.

5. Пройдя ребро , муравей откладывает на нём некоторое количество феромона, которое должно быть связано с оптимальностью сделанного выбора. Обозначим маршрут, пройденный муравьём к моменту времени , как , его длину как , а примем равным порядку длины оптимального пути. Тогда откладываемое количество феромона может быть задано в виде:

Правила внешней среды определяют, в первую очередь, испарение феромона. Пусть есть коэффициент испарения, тогда правило испарения имеет вид , отсюда где под m понимается число муравьев в колонии. В начале алгоритма количества феромона на рёбрах принимается равным небольшому положительному числу. Общее количество муравьёв остаётся постоянным и равным количеству городов, каждый муравей начинает маршрут из своего города.

Программная реализация муравьиного алгоритма представлена в листинге 3.3.

**Листинг 3.3 – Программа для решения задачи коммивояжёра муравьиным алгоритмом**

#include <iostream>

#include <fstream>

#include <cstdlib>

#include <ctime>

#include <time.h>

#include <iomanip>

#include <vector>

#include "Graph.h"

using namespace std;

#define N 5 //количество городов

#define RUNS 50 //число поколений муравьёв

#define alpha 1 //расчётная константа

#define beta 5 //расчётная константа

#define Q 1 //расчётная константа

float minW = FLT\_MAX; //поле памяти, хранящее

//минимальный вес пути

vector <vector <float>> path;

vector <vector <float>> feromon; //массив значений феромонов

vector <float> temp;

**Листинг 3.3 – Программа для решения задачи коммивояжёра муравьиным алгоритмом (продолжение)**

vector <int> instance; //текущий путь

vector <bool> visited;

vector <double> weights;

vector <vector <int>> paths; //все пути

vector <int> bestPath; //наилучший путь

int Search(vector <float> temp, float r)

{

for (int i = 0; i < N; i++)

{

if (temp[i] >= r)

return i;

}

return 1;

}

vector <vector <float>> UpdateFeromon(vector <vector <float>> f, vector <int> instance)

{

for (int k = 0; k < N; k++)

{

for (int i = 0; i < N; i++)

{

f[k][i] \*= 0.95;

}

}

for (int i = 0; i < N - 1; i++)

{

int a = instance[i];

int b = instance[i + 1];

f[a][b] += 0.2;

}

f[instance[N - 1]][instance[0]] += 0.2;

return f;

}

float RoadLength(vector <vector <float>> path, vector <int> currentPath)

{

float weight = 0;

for (int i = 0; i < N - 1; i++)

{

weight += path[currentPath[i]][currentPath[i + 1]];

}

weight += path[currentPath[N - 1]][currentPath[0]];

return weight;

}

vector <vector <float>> ReadMatrix(vector <vector <float>> path)

{

ifstream input("input.txt");

for (size\_t i = 0; i < N; i++) //чтение матрицы смежности

{

for (size\_t j = 0; j < N; j++)

{

input >> path[i][j];

}

}

**Листинг 3.3 – Программа для решения задачи коммивояжёра муравьиным алгоритмом (продолжение)**

input.close();

return path;

}

void MakeFeromonMatrix(vector <vector <float>> feromonMatrix)

{

for (int i = 0; i < N; i++)

{

for (int j = 0; j < N; j++) //чтение матрицы феромонов

{

if (i == j)

feromonMatrix[i][j] = 0.0;

}

}

return;

}

void AntSolverRun()

{

path.clear(); //обновление данных

path.resize(N, vector<float>(N, FLT\_MAX));

feromon.clear();

feromon.resize(N, vector<float>(N, 1));

MakeFeromonMatrix(feromon);

path = ReadMatrix(path);

int sum;

temp.clear(); temp.resize(N, 0);

instance.clear(); instance.resize(N, 0);

visited.clear(); visited.resize(N, false);

int R = rand() % N; //выбор первой вершины

instance[0] = R;

visited[R] = true;

for (int k = 1; k < N; k++)

{

sum = 0;

for (int i = 0; i < N; i++) //расчет феромоновых путей

{

temp[i] = path[R][i] \* feromon[R][i];

sum += path[R][i];

}

for (int i = 0; i < N; i++)

{

temp[i] /= sum;

if (i > 0)

temp[i] += temp[i - 1];

}

while (true) //проход по всем вершинам

{

float r = ((float)rand() / (RAND\_MAX));

int next = Search(temp, r);

if (!visited[next]) //если вершина не посещена

{

visited[next] = true;

**Листинг 3.3 – Программа для решения задачи коммивояжёра муравьиным алгоритмом (продолжение)**

instance[k] = next;

break;

}

}

}

feromon = UpdateFeromon(feromon, instance); //обновление феромоновых

//следов

float weight = RoadLength(path, instance); //расчёт веса текущего пути

if (weight < minW) //отбор наикратчайшего пути

{

minW = weight;

bestPath = instance;

}

return;

}

int main()

{

int i = 0;

do {

AntSolverRun();

i++;

} while (i < RUNS);

for (auto to : bestPath)

cout << to + 1 << " --> ";

cout << bestPath[0]+1;

cout << "\nweight: " << minW;

return 0;

}

### ***Сравнительная характеристика приведённых методов***

Сравнение эффективности использованных алгоритмов проведём на основании времени исполнения соответствующих программ. Замеры времени производятся на одном и том же вычислительном устройстве, одним и тем же способ и на идентичных наборах данных. Время работы программы вычисляется путем вычитания начального системного времени из конечного. Для всех программ системное время засекается с помощью функции clock() заголовочного файла <ctime> на языке программирования С++.

|  |  |  |
| --- | --- | --- |
| Таблица 3.2 – Результаты замеров времени | |  |
| **А) Алгоритм ближайшего соседа** | | |
| Число вершин | Время работы, с | Результат |
| 5 | 0.003 | 43 |
| 10 | 0.004 | 142 |
| 15 | 0.003 | 234 |
| 50 | 0.012 | 1103 |
| 100 | 1.002 | 2141 |
| **Б) Метод ветвей и границ** | | |
| Число вершин | Время работы, с | Результат |
| 5 | 0.012 | 41 |
| 10 | 0.041 | 164 |
| 15 | 0.16 | 234 |
| 50 | 3.54 | 1103 |
| 100 | – | Не найден |
| **В) Муравьиный алгоритм** | | |
| Число вершин | Время работы, с | Результат |
| 5 | 0.006 | 41 |
| 10 | 0.014 | 142 |
| 15 | 0.017 | 234 |
| 50 | 1.32 | 1103 |
| 100 | 3.146 | 2141 |

На основании приведённых тестов, а также по результатам исследований других авторов, можно сделать вывод о том, что для графов с практически полным заполнением матрицы смежности, из-за которого появляется большое число негамильтоновых циклов и повышается сложность решения, наилучшие результаты по точности даёт муравьиный алгоритм. [14]

Известно, что задача коммивояжёра является так называемой NP-трудной задачей (от англ. non-deterministic polynomial), то есть входит в число задача, часто встречающихся на практике, но не имеющих эффективных алгоритмов решения, то есть точные алгоритмы решения этой задаси не могут иметь временную сложность меньше экспоненциальной. [7] Таким образом, на основании полученных данных и теоретических сведениях, можно сделать вывод о сложности рассматриваемых методов решения задачи коммивояжёра (таблица 3.3).

|  |  |
| --- | --- |
| Таблица 3.3 – Сложность рассматриваемых алгоритмов | |
| **Название алгоритма** | **Временная сложность** |
| Алгоритм ближайшего соседа | Факториальная – |
| Метод ветвей и границ | В худшем случае факториальная, в среднем линейно-логарифмическая |
| Муравьиный алгоритм | Квадратичная ) |

# **ЗАКЛЮЧЕНИЕ**

Задача коммивояжёра, известная ещё с 20 века, до сих пор не имеет эффективного решения с точки зрения точности и скорости выполнения. Тем не менее, современные вычислительные мощности позволяют находить оптимальные решения этой задачи для огромных размерностей графов. Более того, многие базовые алгоритмы имеют разные возможности оптимизации. Применение эвристики позволяет оптимизировать решение для частных случаев, рассматриваемых на практике, например, при работе с транспортными сетями и решении транспортных задач.

На основе проведённых исследований можно подвести следующие итоги проделанной работы. В ходе выполнения курсовой работы были рассмотрены такие понятия, связанные с графами и циклами, а также определена постановка задачи коммивояжёра. Программно реализованы такие методы решения, как алгоритм ближайшего соседа, метод ветвей и границ и муравьиный алгоритм. Проведён анализ наиболее рациональных методов решения и определены области их эффективного использования: для малого числа вершин подходит перебор, в частности, алгоритм ближайшего соседа; для большого числа вершин рациональнее применять метод ветвей и границ или иные эвристические методы.

Можно предположить, что ключевым направлением развития технологий в будущем будут являться синергия и уподобление уже существующим, выработанным за миллионы лет эволюции природным механизмам. Многие природные процессы решают задачи выбора оптимального варианта из огромного множества вариантов. Так, на примере муравьиной колонии удалось обнаружить, что переход от классической к синергетической парадигме, то есть к построению системы, способной самостоятельно конструировать метод решения задачи, оказывается эффективнее привычной модели, в которой разработчик интеллектуальных систем сам обязан конструировать для каждой задачи метод решения.

# **БИБЛИОГРАФИЧЕСКИЙ СПИСОК**

1. Домнин П.Н. Элементы теории графов. – Пенза: Изд-во ПГУ, 2004. – 139 с.
2. Кристофидес Н. Теория графов. Алгоритмический подход. – М.: Мир, 1978. – 427 с.
3. Харари Ф., Палмер Э. Перечисление графов. – М.: Мир, 1977. – 324 с.
4. Максимов Д. Пути и маршруты // Наука и жизнь. – 2020. – №2. – С. 81-86.
5. Берж К. Теория графов и ее применения. М.: Изд-во ИЛ, 1962. – 320 с.
6. Гамильтонов цикл и задача коммивояжера // МегаЛекции URL: https://megalektsii.ru/s52787t7.html (дата обращения: 04.11.2022).
7. В.И. Мудров. Задача о коммивояжёре. — М.: «Знание», 1969. — С. 62.
8. Хаггарти Р. Дискретная математика для программистов. –М.: Техно- сфера, 2004. – 320 с.
9. G. Gutin, A. Yeo, A. Zverovich. Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the TSP Архивная копия от 29 июля 2007 на Wayback Machine // Discrete Applied Mathematics 117 (2002)
10. Вишняков П.О. Планирование маршрутов с использованием модифицированного метода «ближайшего соседа» //Математические методы в технике и технологиях - ММТТ. 2014. № 6 (65). С. 63-67.
11. Э. Майника. Алгоритмы оптимизации на сетях и графах. — Москва: Мир, 1981.
12. Галяутдинов Р.Р. Задача коммивояжера - метод ветвей и границ // Сайт преподавателя экономики. [2020]. URL: http://galyautdinov.ru/post/zadacha-kommivoyazhera (дата обращения: 12.11.2022).
13. Штовба С.Д. Муравьиные алгоритмы // Exponenta Pro. Математика в приложениях, 2003, №4, с.70-75.
14. Колесников А.В., Кириков И.А., Листопад С.В., Румовская С.Б., Доманицкий А.А. Решение сложных задач коммивояжера методами функциональных гибридных интеллектуальных систем. - М.: 2011. - 295 с.

# **ПРИЛОЖЕНИЕ 1**

**Листинг 1. Реализация пользовательского класса Graph, предназначенного для работы с графами**

#pragma once

#include <vector>

#include <fstream>

#include <iostream>

#include <stdlib.h>

#include <iomanip>

#include <set>

#include <algorithm>

#include <string>

using namespace std;

class Graph //G = (X, V)

{

public:

int nodeCount = 0; //количество вершин

int edgeCount = 0; //количество рёбер

vector <pair<pair<int, int>, int>> edges;

Graph()

{

nodeCount = 0;

edgeCount = 0;

edges.resize(0);

};

Graph(int n, int m)

{

nodeCount = n;

edgeCount = m;

edges.resize(m+1);

} //конструктор

Graph& operator=(const Graph& A)

{

edgeCount = A.edgeCount;

nodeCount = A.nodeCount;

edges = A.edges;

return \*this;

}

void ReadMatrix();

vector <vector <int>> MatrixToVector();

void ReadListEdgesFromFile();

void ReadListEdgesFromConsole();

void PrintEdges();

void PrintMatrix();

vector <vector <int>> MakeMatrixFromList();

vector <vector <float>> MakeFloatMatrixFromList();

bool IsGamilton()

**Листинг 1. Реализация пользовательского класса Graph, предназначенного для работы с графами (продолжение)**

{

vector <int> degreeList = this->CountVertexDegree();

return count(degreeList.begin(), degreeList.end(), 1) == 0;

}

vector <int> CountVertexDegree()

{

vector <vector <int>> matrix = this->MatrixToVector();

vector <int> degreeList(nodeCount, 0);

for (int i = 0; i < nodeCount; i++)

{

degreeList[i] = nodeCount - count(matrix[i].begin(), matrix[i].end(), 0);

}

return degreeList;

}

~Graph(){};

};

void PrintMatrix(vector <vector <int>> matrix, string diagonalSymbol)

{

cout << "\n";

for (int i = 0; i < matrix.size(); i++)

{

cout << "\t\t\t\t\t\t";

for (int j = 0; j < matrix[i].size(); j++)

{

if (i != j)

printf("%\*d", 4, matrix[i][j]);

else printf("%\*s", 4, diagonalSymbol.c\_str());

} std::cout << "\n";

}

return;

}

void PrintMatrix(vector <vector <int>> matrix)

{

cout << "\n\n";

for (int i = 0; i < matrix.size(); i++)

{

cout << "\t\t\t\t\t\t";

for (int j = 0; j < matrix[i].size(); j++)

{

if (i != j)

printf("%\*d", 4, matrix[i][j]);

else printf("%\*d", 4, -1);

} std::cout << "\n";

}

return;

}

void PrintCuttedMatrix(vector <vector <int>> matrix, int row, int column)

{

cout << "\n\n";

for (int i = 0; i < matrix.size(); i++)

{

cout << "\t\t\t\t\t\t";

for (int j = 0; j < matrix[i].size(); j++)

{

if (i != j && i != row && j != column)

printf("%\*d", 4, matrix[i][j]);

**Листинг 1. Реализация пользовательского класса Graph, предназначенного для работы с графами (продолжение)**

if (i == j && i != row && j != column)

printf("%\*s", 4, "inf");

} std::cout << "\n";

}

return;

}

void Graph::PrintMatrix()

{

system("cls");

vector <vector <int>> matrix(nodeCount, vector<int>(nodeCount, 0));

matrix = this->MatrixToVector();

cout << "\n\t\t\t\tМатрица смежности:\n\n";

for (int i = 0; i < this->nodeCount; i++)

{

cout << "\t\t\t\t\t\t";

for (int j = 0; j < this->nodeCount; j++)

{

cout << matrix[i][j] << " ";

}

cout << "\n";

}

return;

}

void Graph::PrintEdges()

{

system("cls");

int i = 0;

cout << "\n";

cout << "\n\t\t\t\tСписок рёбер:\n";

for (auto to : this->edges)

{

i++;

cout << "\t\t\t\tРебро " << i << ". " << "(" << to.first.first + 1

<< ", " << to.first.second + 1 << ")" << " с весом " << to.second

<< "\n";

}

}

void Graph::ReadMatrix()

{

ifstream input("input.txt");

int n, m = 0, a;

input >> n;

vector <vector <int>> matrix(n, vector<int>(n, 0));

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

{

input >> a;

if (a == 1) m++;

matrix[i][j] = a;

}

}

this->nodeCount = n;

this->edgeCount = m / 2;

this->edges.clear();

for (int i = 0; i < n; i++)

{

**Листинг 1. Реализация пользовательского класса Graph, предназначенного для работы с графами (продолжение)**

for (int j = 0; j < n; j++)

{

if (matrix[i][j] != 0)

{

pair <pair<int, int>, int> vertexWithWeight;

vertexWithWeight = make\_pair(make\_pair(i + 1, j + 1), matrix[i][j]);

this->edges.push\_back(vertexWithWeight);

}

}

}

sort(this->edges.begin(), this->edges.end());

input.close();

system("cls");

cout << "\n\n\t\t\t\t!!!! Матрица смежности успешно считана!\n\n";

return;

}

void Graph::ReadListEdgesFromFile()

{

ifstream input("input.txt");

int n, m;

input >> n;

input >> m;

this->edges.clear();

for (int i = 0; i < m; i++)

{

int a, b, weight;

input >> a >> b >> weight;

pair <pair<int, int>, int> vertexWithWeight = make\_pair(make\_pair(a, b), weight);

edges.push\_back(vertexWithWeight);

}

input.close();

system("cls");

if (!this->edges.empty()) cout << "\n\n\t\t\t\t!!!! Список рёбер успешно считан!\n\n";

return;

}

void Graph::ReadListEdgesFromConsole()

{

int n, m;

cout << "\n";

cout << "\n\t\t\t\tВведите число вершин: "; cin >> n;

cout << "\n\t\t\t\tВведите число рёбер: "; cin >> m;

this->edges.clear();

cout << "\n\t\t\t\tВведите список рёбер:\n";

for (int i = 0; i < m; i++)

{

int a, b;

int weight;

cout << "\t\t\t\tРебро " << i + 1 << ". "; cin >> a >> b;

cout << "\t\t\t\tЕго вес: "; cin >> weight;

pair <pair<int, int>, int> vertexWithWeight = make\_pair(make\_pair(a, b), weight);

edges.push\_back(vertexWithWeight);

}

return;

}

vector <vector <int>> Graph::MatrixToVector()

**Листинг 1. Реализация пользовательского класса Graph, предназначенного для работы с графами (продолжение)**

{

int inf = INT\_MAX; //значение, принимаемое за бесконечность

vector <vector <int>> matrix(nodeCount, vector <int>(nodeCount, 0));

int i, j;

for (auto to : edges)

{

i = to.first.first - 1;

j = to.first.second - 1;

matrix[i][j] = to.second;

}

for (int i = 0; i < nodeCount; i++)

{

for (int j = 0; j < nodeCount; j++)

{

if (i == j) matrix[i][j] = inf;

}

}

return matrix;

};

vector <vector <int>> Graph::MakeMatrixFromList()

{

vector <vector <int>> matrix(this->nodeCount, vector<int>(this->nodeCount, 0));

for (int i = 0; i < this->edges.size(); i++)

{

matrix[this->edges[i].first.first - 1][this->edges[i].first.second - 1] = this->edges[i].second;

}

return matrix;

}