

МИНОБРНАУКИ РОССИИ  
Федеральное государственное бюджетное  
образовательное учреждение высшего образования  
«Тульский государственный университет»  
Кафедра информационной безопасности

УТВЕРЖДАЮ

Зав. кафедрой ИБ

\_\_\_\_\_ А.А. Сычугов

« \_\_\_\_ » \_\_\_\_\_ 20 \_\_\_\_ г.

## ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовой работе по дисциплине  
«ДИСКРЕТНАЯ МАТЕМАТИКА»  
на тему

«Гамильтоновы циклы и задача коммивояжёра»

Автор работы \_\_\_\_\_ студент гр. 230711 Павлова В.С.  
(дата, подпись) (фамилия и инициалы)

Руководитель работы \_\_\_\_\_  
(дата, подпись) (должность) (фамилия и инициалы)

Работа защищена \_\_\_\_\_ с оценкой \_\_\_\_\_  
(дата)

Члены комиссии \_\_\_\_\_  
(дата, подпись) (должность) (фамилия и инициалы)

\_\_\_\_\_ (дата, подпись) (должность) (фамилия и инициалы)

\_\_\_\_\_ (дата, подпись) (должность) (фамилия и инициалы)

Тула 20 22

УТВЕРЖДАЮ

Зав. кафедрой ИБ

А.А. Сычугов

«\_\_\_» \_\_\_\_\_ 20\_\_ г.

## ЗАДАНИЕ

на курсовую работу по дисциплине

«ДИСКРЕТНАЯ МАТЕМАТИКА»

студенту гр. 230711 Павловой Виктории Сергеевне

(фамилия, имя, отчество)

Тема работы

«Гамильтоновы циклы и задача коммивояжёра»

Входные данные

1. В.И. Мудров. Задача о коммивояжёре. — М.: «Знание», 1969. — С. 62.

2. Хаггарти Р. Дискретная математика для программистов. — М.: Техно-сфера, 2004. — 320 с.

Задание получил

(подпись)

(дата)

График выполнения работы

1 неделя – Получение и ознакомление с заданием

2–4 неделя – Изучение литературы и других исходных материалов

5–10 неделя – Изучение теории, раскрывающей тему курсовой работы

11–12 неделя – Разработка программной реализации курсовой работы

13 неделя – Анализ результатов

14–15 неделя – Оформление пояснительной записки и сдача на проверку

16 неделя – Защита курсовой работы

Замечания консультанта

К защите. Консультант работы

(подпись)

(дата)

## **ОГЛАВЛЕНИЕ**

<b>ВВЕДЕНИЕ</b> .....	4
<b>I. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ</b> .....	5
1.1. Гамильтоновы циклы и графы .....	5
1.2. Задача о поиске минимального гамильтонова цикла .....	6
<b>II. ПРАКТИЧЕСКАЯ ЧАСТЬ</b> .....	8
2.1. Постановка задачи коммивояжёра .....	8
2.2. Методы и алгоритмы решения задачи коммивояжёра .....	9
<b>ЗАКЛЮЧЕНИЕ</b> .....	27
<b>БИБЛИОГРАФИЧЕСКИЙ СПИСОК</b> .....	28
<b>ПРИЛОЖЕНИЕ</b> .....	29

## 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 указанной задачей, а в качестве предмета исследования – алгоритм нахождения гамильтонова цикла на примере задачи о коммивояжёре.

Целью курсовой работы является изучение и применение алгоритмов нахождения гамильтонового цикла минимального веса на примере конкретной задачи, где под весом цикла понимается сумма весов составляющих его ребер.

# І. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

## 1.1. Гамильтоновы циклы и графы

Путем или цепью в графе называют конечную последовательность вершин, в которой каждая вершина (кроме последней) соединена со следующей в последовательности вершин ребром. 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*.

Граф, который содержит простой путь, проходящий через каждую его вершину, называется *полугамильтоновым*. Это определение можно распространить на ориентированные графы, если путь считать ориентированным. [1]

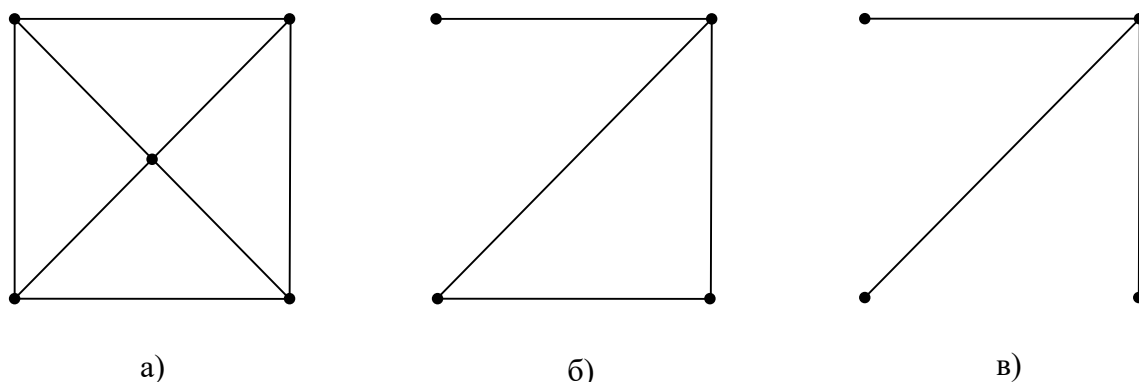
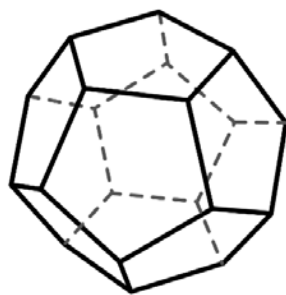
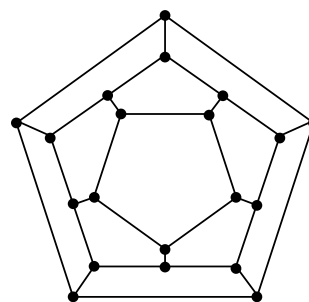


Рис. 1.1 – Гамильтонов граф (а), полугамильтонов граф (б) и негамильтонов граф (в)

Гамильтоновы путь, цикл и граф названы в честь ирландского математика Уильяма Гамильтона, который занимался исследованием задачи «кругосветного путешествия» по додекаэдру (рисунок 1.2, 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 1.2, б). [2]



а)



б)

Рис. 1.2 – 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 существования гамильтонова цикла в графе можно сформулировать так: если неориентированный граф  $G$  содержит гамильтонов цикл, тогда в нём не существует ни одной вершины  $x_i$  с локальной степенью  $p(x_i) < 2$ . Под локальной степенью вершины понимается число рёбер, ей инцидентных. [3]

### ***1.2. Задача о поиске минимального гамильтонова цикла***

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. [4]

Для решения этой задачи существует простой алгоритм – полный перебор всех возможных вариантов. Однако, такой подход, как правило, неприемлем из-за сложности порядка  $O(n!)$ . 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. [4]

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  $(x_i, x_j)$  одинаковы расстояния  $r_{ij} = r_{ji}$ . 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  $(x_i, x_j)$  никогда не бывает длиннее через промежуточную вершину  $x_k$ . [5]

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.

## II. ПРАКТИЧЕСКАЯ ЧАСТЬ

### 2.1. Постановка задачи коммивояжёра

Конкретизируем условия задачи о поиске гамильтонова цикла минимального веса для того, чтобы можно было провести корректный сравнительный анализ различных методов решения. Возьмём пять городов Тульской области, связанных между собой дорогами: Тула, Новомосковск, Ефремов, Белёв и Щёкино. Для простоты примем, что каждый из городов имеет ровно одну дорогу во все остальные, то есть в каждый город входит и из каждого города исходит по четыре дороги. Междугородние шоссе имеют разное состояние изношенности, следовательно проезд по ним характеризуется разными условными единицами «стоимости».

Необходимо составить наиболее выгодный (то есть минимальный по стоимости) маршрут туристического автобуса для посещения всех пяти городов. Таким образом, необходимо найти гамильтонов цикл наименьшего веса для полного ориентированного взвешенного графа  $G$ , заданного следующей матрицей смежности:

Таблица 2.1 – Матрица смежности  $S$  графа  $G$

	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$
$x_1$	0	20	18	12	8
$x_2$	5	0	14	7	11
$x_3$	12	18	0	6	11
$x_4$	11	17	11	0	12
$x_5$	5	5	5	5	0



## 2.2. Методы и алгоритмы решения задачи коммивояжёра

Для проведения объективного сравнения результатов рассмотрим следующие три метода решения задачи коммивояжёра:

- а) алгоритм ближайшего соседа (жадный алгоритм);
- б) метод ветвей и границ;
- в) муравьиный алгоритм.

### *Алгоритм ближайшего соседа*

Как было сказано ранее, эффективный алгоритм решения задачи коммивояжёра до сих пор 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. Brutforce) сложность равна  $O(n!)$ . 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. [6]

Идея данного алгоритма основана на следующем условии: если на каждом шаге идти в ближайшую вершину, то конечный маршрут получится довольно хорош в целом. Перед туристическим автобусом 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. [7]

Возьмём в качестве начальной вершины  $x_1$ . Ближайшая к ней вершина –  $x_5$ , расстояние до неё – 8. Она равноудалена от всех остальных вершин, значит переместимся в неё и выберем любую из них. В первом городе мы уже были, выберем второй. Далее логично было бы отправиться из второго города в четвёртый, расстояние до которого 7, однако данный путь не приведёт нас к гамильтонову циклу, поэтому единственным подходящим вариантом будет вершина 3. Из неё мы отправимся по дороге длиной 6 в город  $x_4$  и, наконец, замкнём цикл. Итого общий вес пути:  $8 + 5 + 14 + 6 + 11 = 44$ . Таким образом найденный гамильтонов цикл имеет вид:  $1 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1$ .

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. [8] Смысл данного алгоритма выражен блок-схемой, представленной на рисунке 2.1.

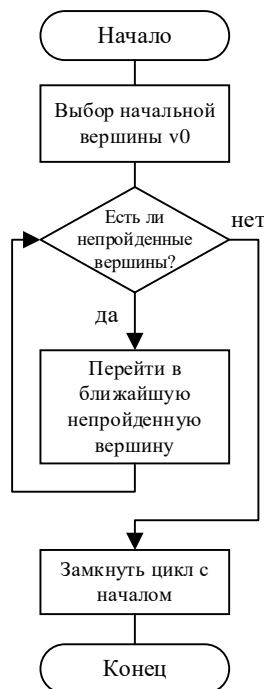


Рис. 2.1 – Блок-схема алгоритма ближайшего соседа

Если перебирать все возможные варианты путей в графах, становится ясно, что для него существует несколько гамильтоновых циклов разного веса. 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. [7]

Программная реализация данного алгоритма представлена в листинге 1.

### Листинг 1 – Программа для решения задачи коммивояжёра методом ближайшего соседа

```
#include <Windows.h>
#include <iostream>
#include "Graph.h"
#include <vector>
#include <fstream>
using namespace std;

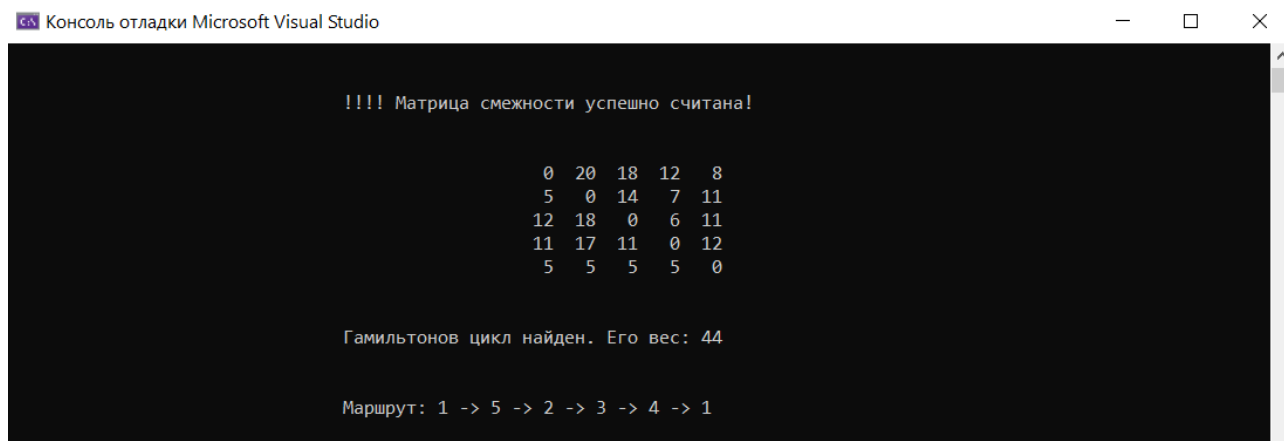
int weight = 0; //поле для хранения веса маршрута
vector <bool> visited; //массив для помечания пройденных вершин
vector <int> path; //массив, хранящий найденный путь
int FindGamiltonWithWeight(vector <vector <int>> matrix, int cur)
{
    path.push_back(cur); //пометили начальную вершину

    if (path.size() == matrix.size()) //если пройдены все вершины
    {
        if (matrix[path.back()][path[0]] != 0) { //если цикл замкнут
            weight += matrix[path.back()][path[0]]; //внести вес последнего ребра
            path.push_back(path[0]); //пометить последнюю вершину
            return weight; //вернуть вес найденного пути
        }
        else {
            path.pop_back(); //отметить вершину непройденной
            return -1;
        }
    }
    visited[cur] = true; //отметить вершину пройденной
    int i = 0; int k = 0;
    vector <int> local;
    while (i < matrix.size()-1)
    {
        local = matrix[cur];
        sort(local.begin(), local.end());
        k = distance(matrix[cur].begin(), find(matrix[cur].begin(), matrix[cur].end(),
        local[i]));
        if (k == 0) k = 1;
```

## Листинг 1 – Программа для решения задачи коммивояжёра методом ближайшего соседа (продолжение)

```
//если есть ребро и следующая вершина не была пройдена
if (!visited[k] && k != cur)
{
    //внести вес новой пройденной вершины в общий вес пути
    weight += matrix[cur][k];
    //запуск алгоритма из новой вершины
    if (FindGamiltonWithWeight(matrix, k) != -1)
    {
        return weight;
    }
    else weight -= matrix[cur][k];
}
i++;
}
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();
    vector<vector<int>> matrix = graph.MakeMatrixFromList();
    for (size_t i = 0; i < matrix.size(); i++)
    {
        for (size_t j = 0; j < matrix[i].size(); j++)
        {
            if (i == j) matrix[i][j] = INT_MAX;
        }
    }
    PrintMatrix(matrix, "0");
    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(matrix, 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\tГамильтонов цикл найден. Его вес: " << minWeight << "\n";
    int i = 0;
    cout << "\n\n\t\t\tМаршрут: ";
    for (auto to : bestPath)
    {
        cout << to + 1;
        if (i < bestPath.size() - 1) cout << " -> ";
        i++;
    }
    cout << "\n\n"; return 0;
}
```

Результат решения задачи коммивояжёра для рассматриваемых условий в данной программной реализации приведён на рисунке 2. 2.



```
Консоль отладки Microsoft Visual Studio

!!!! Матрица смежности успешно считана!

      0  20  18  12  8
      5   0  14   7  11
     12  18   0   6  11
     11  17  11   0  12
      5   5   5   5   0

Гамильтонов цикл найден. Его вес: 44

Маршрут: 1 -> 5 -> 2 -> 3 -> 4 -> 1
```

Рис. 2.2 – Результат работы программы

### *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 выбрав маршрут с наименьшей длиной. [5]

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. 2.3), корнем которого является множество всех решений, а листьями – гамильтоновы циклы.

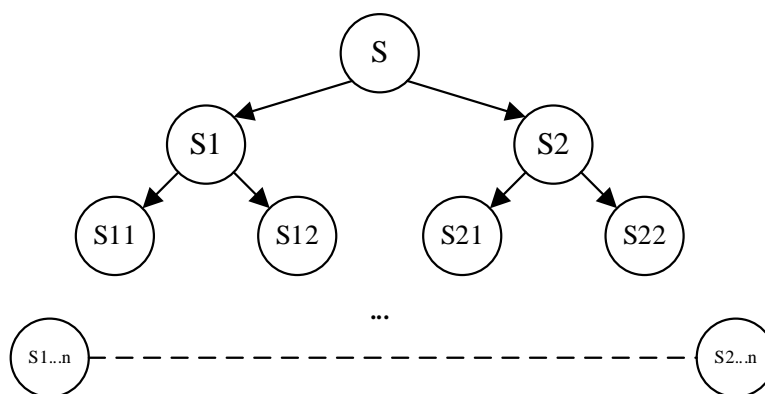


Рис. 2.3 – Дерево решений метода ветвей и границ

Использование понятия границ и их расчёт стимулирует или тормозит рост ветвей в таком дереве. [5] 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*. [9]

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. [9]

Рассмотрим матрицу смежности  $S$  заданного графа. На главной диагонали матрицы помещены прочерки, поскольку диагональные клетки не рассматриваются (петель нет).

	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$
$x_1$	-	20	18	12	8
$x_2$	5	-	14	7	11
$x_3$	12	18	-	6	11
$x_4$	11	17	11	-	12
$x_5$	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). [10]

- 1) Найдём минимум  $m_i$  для каждой из *строк* и произведём редуцирование каждого элемента строки  $S_i$  матрицы  $S$  следующим образом: если  $S_i$  не лежит на диагонали, то  $S_{ij} = S_{ij} - m_i$
- 2) Найдём минимум  $m_j$  для каждого из *столбцов* и произведём редуцирование для каждого элемента столбца  $S_j$  матрицы  $S$  следующим образом: если  $S_{ji}$  не лежит на диагонали, то  $S_{ji} = S_{ji} - m_j$
- 3) Вычислим нижнюю границу в корневой точке решения, как сумму найденных ранее минимумов.

	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	<b>min</b>
$x_1$	-	12	10	4	0	<b>8</b>
$x_2$	0	-	9	2	6	<b>5</b>
$x_3$	6	12	-	0	5	<b>6</b>
$x_4$	0	6	0	-	1	<b>11</b>
$x_5$	0	0	0	0	-	<b>5</b>
<b>min</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	

Рисунок 2.4 – Преобразование матрицы смежности

Таким образом, нижняя граница  $L_b$  корневого решения будет равна следующей сумме  $L_b = 8 + 5 + 6 + 11 + 5 + 0 = 35$ . Прodelывая аналогичные действия для преобразованной матрицы, получим следующее дерево решений (рисунок 2.5):

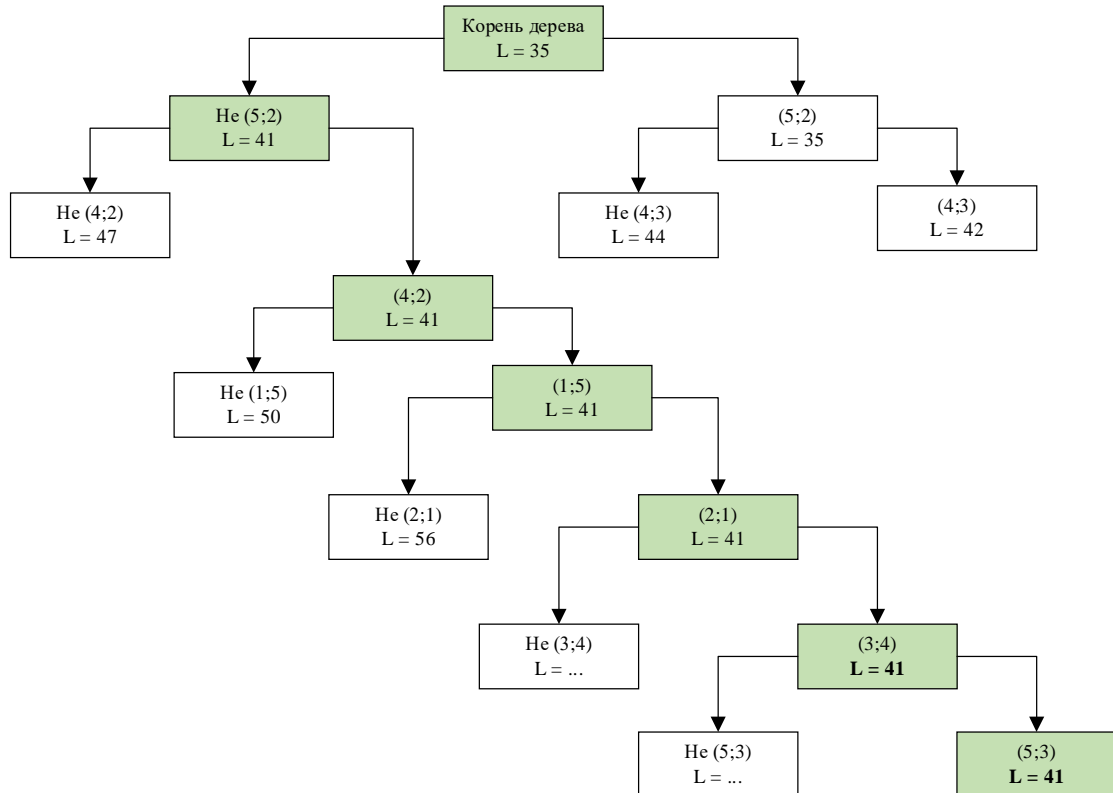


Рис. 2.5 – Дерево решений для графа G

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 2.

## Листинг 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;

```



## Листинг 2 – Программа для решения задачи коммивояжёра методом ветвей и границ (продолжение)

```
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]);
    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
```

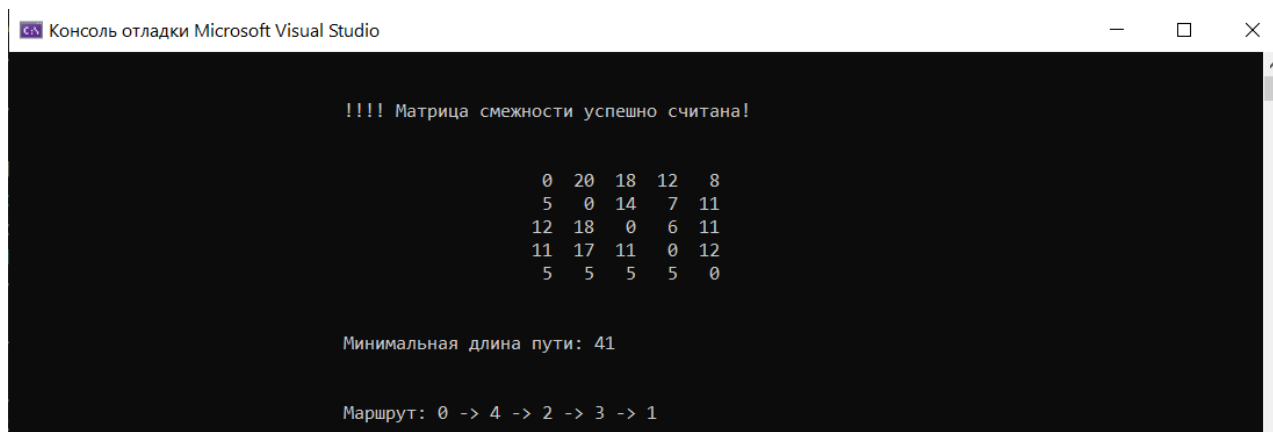
## Листинг 2 – Программа для решения задачи коммивояжёра методом ветвей и границ (продолжение)

```
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]) +
            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();
    PrintMatrix(matrix, "0");
    BranchAndBoundSolver(matrix);
    printf("\n\n\t\t\t\t\tМинимальная длина пути: %d\n", minWeight);
    printf("\n\n\t\t\t\t\tМаршрут: ");
    for (int i = 0; i < graph.nodeCount-1; i++)
        printf("%d -> ", bestPath[i]);
    printf("%d\n", bestPath[matrix.size()-1]); return 0;}
```

Результат решения задачи коммивояжёра для рассматриваемых условий в данной программной реализации приведён на рисунке 2.6.



```
Консоль отладки Microsoft Visual Studio

!!!! Матрица смежности успешно считана!

      0  20  18  12  8
      5   0  14   7  11
      12  18   0   6  11
      11  17  11   0  12
      5   5   5   5   0

Минимальная длина пути: 41

Маршрут: 0 -> 4 -> 2 -> 3 -> 1
```

Рис. 2.6 – Результат работы программы

### *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. [11]

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. [11]

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  $J_{i,k}$  – список городов, которые необходимо посетить муравью  $k$ , находящемуся в городе  $i$ .

2. Муравьи обладают «зрением» – у них есть эвристическое «желание» посетить город  $j$ , если они находятся в городе  $i$ . Принято считать, что видимость обратно пропорциональна расстоянию между городами  $\eta_{ij} = \frac{1}{d_{ij}}$ ;

3. Муравьи обладают «обонянием» – они могут улавливать след феромона, подтверждающий желание посетить город  $j$  из города  $i$  на основании опыта других муравьёв. Количество феромона на ребре  $(i, j)$  в момент времени  $t$  обозначается как  $\tau_{ij}(t)$ .

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

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

Где  $\alpha, \beta$  – параметры, задающие веса следа феромона.

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

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

Правила внешней среды определяют, в первую очередь, испарение феромона. Пусть есть коэффициент испарения, тогда правило испарения имеет вид  $\tau_{ij}(t+1) = |(1-p)\tau_{ij}(t) + \Delta\tau_{ij}(t)|$ , отсюда  $\Delta\tau_{ij}(t) = \sum_{k=1}^m \Delta\tau_{ij,k}(t)$ , 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.

### Листинг 3 – Программа для решения задачи коммивояжёра муравьиным алгоритмом

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <ctime>
#include <time.h>
#include <iomanip>
```

### Листинг 3 – Программа для решения задачи коммивояжёра муравьиным алгоритмом (продолжение)

```
#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;
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]];
}
```

### Листинг 3 – Программа для решения задачи коммивояжёра муравьиным алгоритмом (продолжение)

```
        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];
        }
    }
    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;
        }
    }
}
```

### Листинг 3 – Программа для решения задачи коммивояжёра муравьиным алгоритмом (продолжение)

```
        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;
            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);

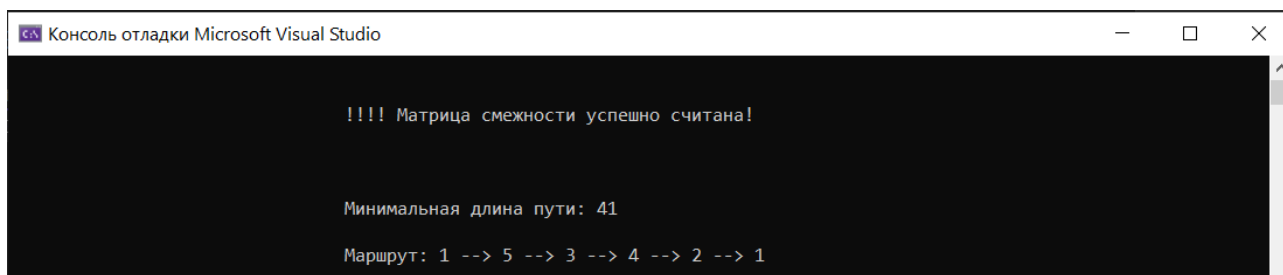
    for (auto to : bestPath)
        cout << to + 1 << " --> ";

    cout << bestPath[0]+1;
    cout << "\nweight: " << minW;

    return 0;
}
```

Результат решения задачи коммивояжёра для рассматриваемых условий в данной программной реализации приведён на рисунке 2.7.



The image shows a screenshot of the 'Консоль отладки Microsoft Visual Studio' (Microsoft Visual Studio Debug Console) window. The window has a dark background and a light-colored title bar. The output text is as follows:

```
!!!! Матрица смежности успешно считана!  
  
Минимальная длина пути: 41  
Маршрут: 1 --> 5 --> 3 --> 4 --> 2 --> 1
```

Рис. 2.7 – Результат работы программы

### *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 идентичном наборе входных данных, соответствующих поставленной задаче. 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 C++.

Таблица 3.2 – Результаты замеров времени

Алгоритм ближайшего соседа		
Название алгоритма	Время работы, с	Результат
Алгоритм ближайшего соседа	0.003	42
Метод ветвей и границ	0.012	41
Муравьиный алгоритм	0.006	41

На основании приведённых тестов, а также по результатам других исследований 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. [12]

Известно, что задача коммивояжёра является так называемой NP-трудной задачей (от англ. non-deterministic polynomial), то есть входит в число задач, часто встречающихся на практике, но не имеющих эффективных алгоритмов решения, то есть точные алгоритмы решения этой задачи не могут иметь временную сложность меньше экспоненциальной. [5]

Таким образом, на основании полученных данных и теоретических сведениях, можно сделать следующий вывод о сложности рассматриваемых методов решения задачи коммивояжёра (таблица 2.3):

Таблица 2.3 – Сложность рассматриваемых алгоритмов

Название алгоритма	Временная сложность
Алгоритм ближайшего соседа	Факториальная – $O(n!)$
Метод ветвей и границ	В худшем случае факториальная $O(n!)$ , в среднем линейно-логарифмическая $O(n * \log_2 n)$
Муравьиный алгоритм	Квадратичная $O(n^2)$

## ЗАКЛЮЧЕНИЕ

Задача коммивояжёра, известная ещё с двадцатого 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 основные понятия, связанные с графами и циклами, а также поставлена задача коммивояжёра. Программно реализованы такие методы решения указанной задачи, 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. Харари Ф., Палмер Э. Перечисление графов. – М.: Мир, 1977. – 324 с.
2. Максимов Д. Пути и маршруты // Наука и жизнь. – 2020. – №2. – С. 81-86.
3. Берж К. Теория графов и ее применения. М.: Изд-во ИЛ, 1962. – 320 с.
4. Гамильтонов цикл и задача коммивояжера // MegaLektsii URL: <https://megalektsii.ru/s52787t7.html> (дата обращения: 04.11.2022).
5. В.И. Мудров. Задача о коммивояжёре. — М.: «Знание», 1969. – С. 62.
6. Хаггарти Р. Дискретная математика для программистов. –М.: Техно- сфера, 2004. – 320 с.
7. G. Gutin, A. Yeo, A. Zverovich. Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the TSP // Discrete Applied Mathematics 117 (2002) – 5 с.
8. Вишняков П.О. Планирование маршрутов с использованием модифицированного метода «ближайшего соседа» // Математические методы в технике и технологиях. – 2014. – №6(65). – С. 63-67.
9. Э. Майника. Алгоритмы оптимизации на сетях и графах. – Москва: Мир, 1981. – 320 с.
10. 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).
11. Штовба С.Д. Муравьиные алгоритмы // Exponenta Pro. Математика в приложениях. – 2003. – №4. – С.70-75.
12. 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.

## 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\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\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\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";
        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\tРёбро " << 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\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\t!!!! Список рёбер успешно
считан!\n\n";
    return;
}

void Graph::ReadListEdgesFromConsole()
{
    int n, m;
    cout << "\n";
    cout << "\n\t\t\t\t\tВведите число вершин: "; cin >> n;
    cout << "\n\t\t\t\t\tВведите число рёбер: "; cin >> m;
    this->edges.clear();
    cout << "\n\t\t\t\t\tВведите список рёбер:\n";
    for (int i = 0; i < m; i++)
    {
        int a, b;
        int weight;
        cout << "\t\t\t\t\tРёбро " << i + 1 << ". "; cin >> a >> b;
        cout << "\t\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;
}
```