Выполнил:

студент группы 19ВВ2

Петров О.М.

Принял:

д.т.н Митрохин М.А.

к.т.н Юрова О. В.

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

К курсовому проектированию

по курсу «Логика и основы алгоритмизации в инженерных задачах»

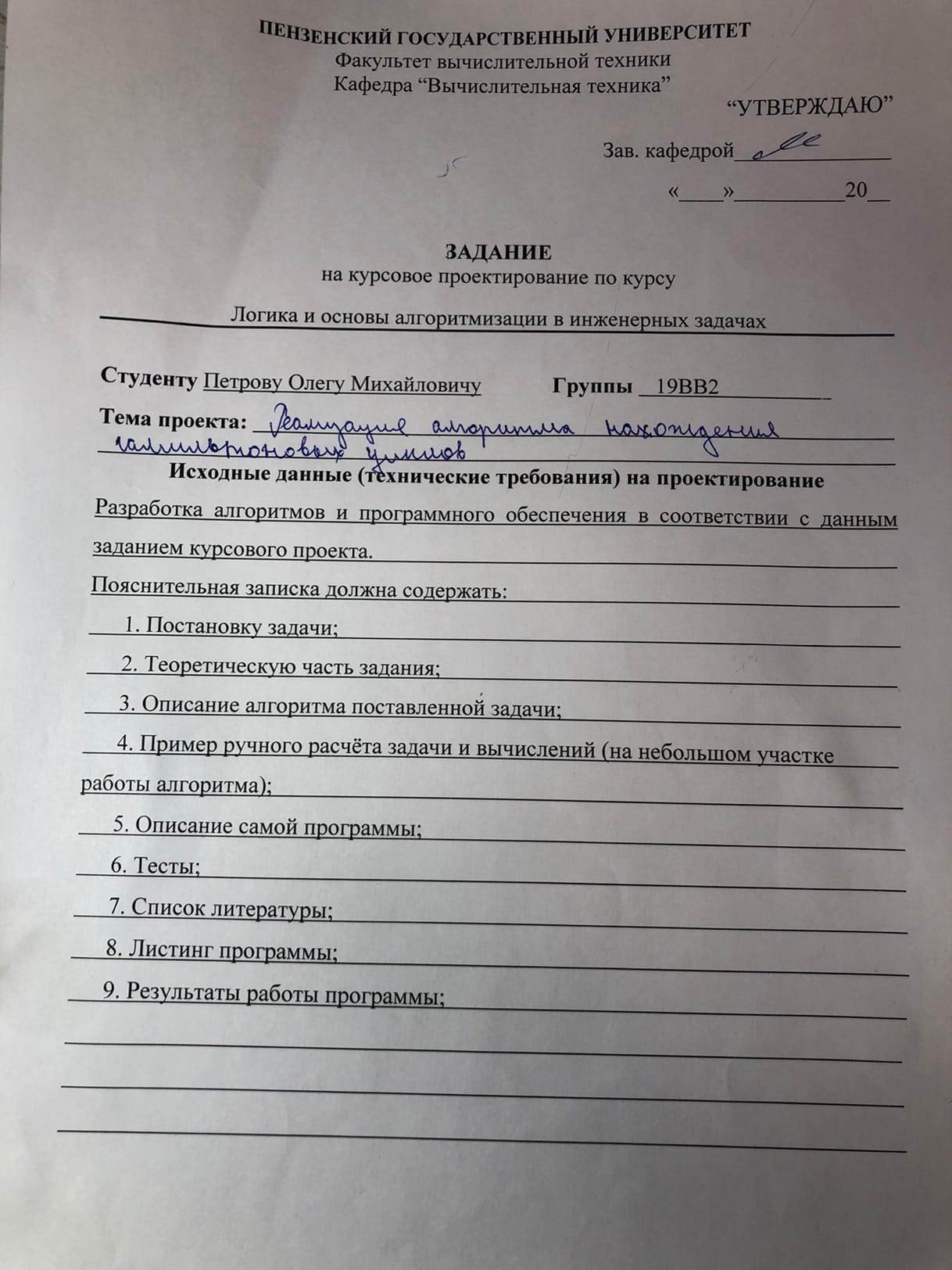
на тему «Реализация алгоритма нахождения Гамильтоновых циклов»

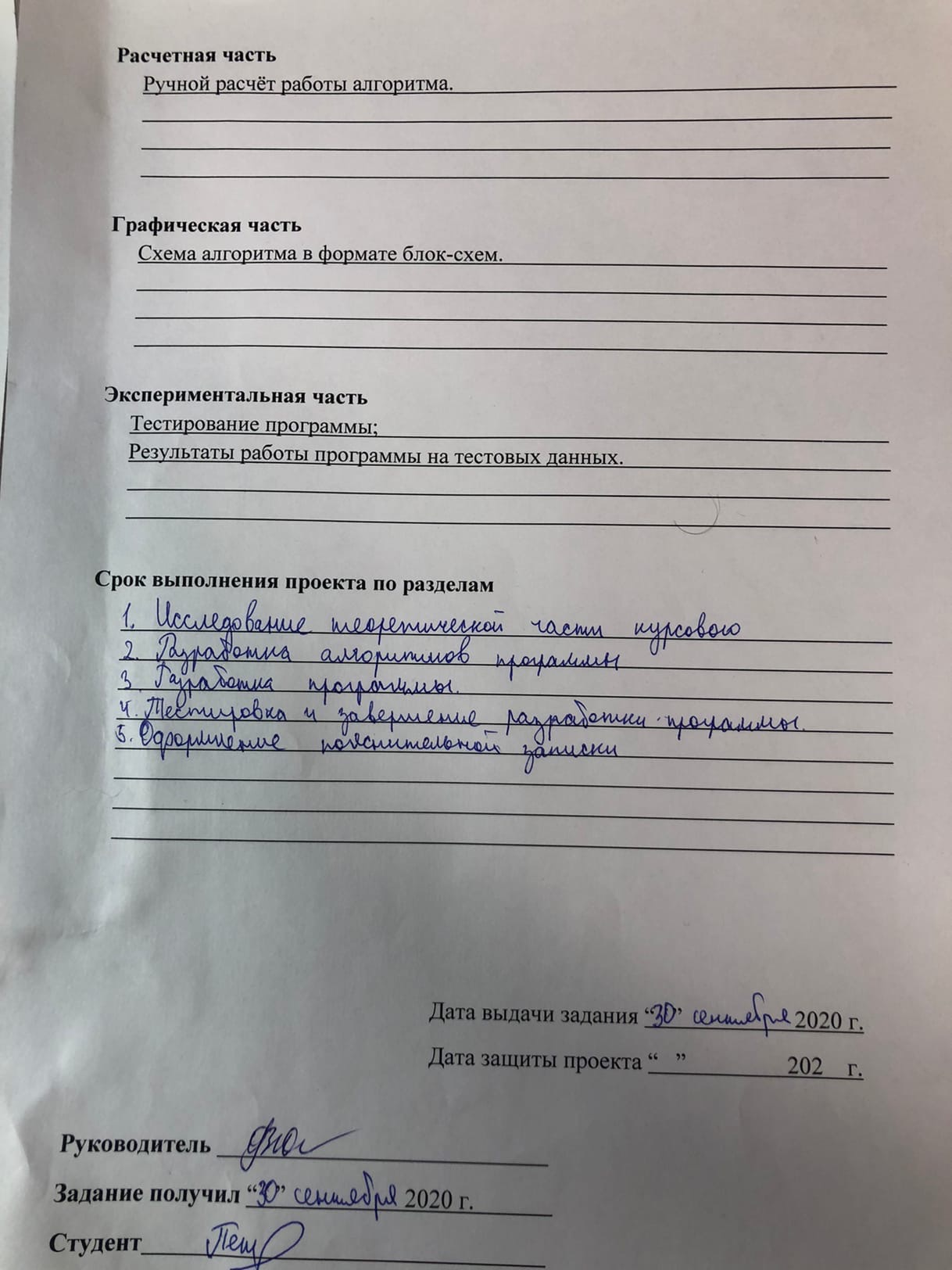
МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Пензенский государственный университет

Кафедра «Вычислительная техника»

Пенза 2020





Содержание

Реферат 6

Введение 7

1. Постановка задачи 8

2. Теоретическая часть9

3. Описание алгоритма поставленной задачи 13

4. Ручной расчёт15

5. Описание программы. 16

6. Тестирование 28

Заключение 34

Список используемых источников 35

Приложение 36

# 

# Реферат

Отчет 46 стр., 20 рисунков и 1 таблица.

ГРАФ, ТЕОРИЯ ГРАФОВ, ОРГРАФ, АЛГОРИТМ НАХОЖДЕНИЯ ГАМИЛЬТОНОВЫХ ЦИКЛОВ, ПОИСК КРАТЧАЙШИХ ГАМИЛТОНОВЫХ ЦИКЛОВ.

Цель исследования – разработка программы, которая будет находить все замкнутые пути, которые проходят через каждую вершину графа ровно по одному разу, используя алгоритм нахождения Гамильтоновых циклов.

# Введение

Темой данного курсового проекта является алгоритм поиска Гамильтоновых циклов в графе.

Рассматриваемая задача является одной из самых ста­рей­ших в теории графов, она также имеет реальное историческое начало.

Гамильтоновы путь, цикл и граф названы в честь ирландского математика Уильям Гамильтон (Hamilton W.), который В 1859 г. впервые определил эти классы, исследовав задачу «кругосветного путешествия» по [додекаэдру](https://ru.wikipedia.org/wiki/%D0%94%D0%BE%D0%B4%D0%B5%D0%BA%D0%B0%D1%8D%D0%B4%D1%80). В этой задаче вершины додекаэдра символизировали известные города,такие\_как [Брюссель](https://ru.wikipedia.org/wiki/%D0%91%D1%80%D1%8E%D1%81%D1%81%D0%B5%D0%BB%D1%8C), [Амстердам](https://ru.wikipedia.org/wiki/%D0%90%D0%BC%D1%81%D1%82%D0%B5%D1%80%D0%B4%D0%B0%D0%BC), [Эдинбург](https://ru.wikipedia.org/wiki/%D0%AD%D0%B4%D0%B8%D0%BD%D0%B1%D1%83%D1%80%D0%B3), [Пекин](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D0%BA%D0%B8%D0%BD), [Прага](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B0%D0%B3%D0%B0), [Дели](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BB%D0%B8), [Франкфурт](https://ru.wikipedia.org/wiki/%D0%A4%D1%80%D0%B0%D0%BD%D0%BA%D1%84%D1%83%D1%80%D1%82) и др., а рёбра — соединяющие их дороги. Путешествующий должен пройти «вокруг света», найдя путь, который проходит через все вершины ровно один раз[[3]](https://ru.wikipedia.org/wiki/%D0%93%D0%B0%D0%BC%D0%B8%D0%BB%D1%8C%D1%82%D0%BE%D0%BD%D0%BE%D0%B2_%D0%B3%D1%80%D0%B0%D1%84#cite_note-_532c29f84d9e2750-3). Чтобы сделать задачу более интересной, порядок прохождения городов устанавливался заранее. А чтобы было легче запомнить, какие города уже соединены, в каждую вершину додекаэдра был вбит гвоздь, и проложенный путь отмечался небольшой верёвкой, которая могла обматываться вокруг гвоздя. Однако такая конструкция оказалась слишком громоздкой, и Гамильтон предложил новый вариант игры, заменив додекаэдр плоским графом, [изоморфным](https://ru.wikipedia.org/wiki/%D0%98%D0%B7%D0%BE%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC_%D0%B3%D1%80%D0%B0%D1%84%D0%BE%D0%B2) графу, построенному на рёбрах додекаэдра

В качестве среды разработки мною была выбрана среда MicrosoftVisualStudio2019, языки программирования – C и C++.

Целью данной курсовой работы является разработка программы на языках C и C++, которые является широко используемыми. Именно с их помощью в данном курсовом проекте реализуется алгоритм поиска Гамильтоновых циклов.

# 1. Постановка задачи

Целью данной работы является разработка программы, которая сможет анализировать граф, заданный матрицей смежности, для поиска Гамильтоновых циклов в графе.

Программа должна работать следующим образом:

Пользователь вводит количество вершин матрицы смежности.

Далее программа предоставит возможность заполнения матрицы смежности автоматически и вручную. После обработки этих данных на экран должна выводиться матрица смежности.

Следующим шагом работы программы, она должна сделать проверку графа на Гамильтоновы циклы. Если граф удовлетворяет условиям, то должны быть построены Гамильтоновы циклы и результаты работы записаны в файл. Иначе пользователю будет предложено заполнить матрицу смежности заново или завершить работу с программой.

Устройство ввода – клавиатура, проект консольный.

# 2. Теоретическая часть

Граф G (рисунок 1) задается множеством вершин V1, V2, ..., Vn. и множеством ребер , соединяющих между собой определенные вершины.

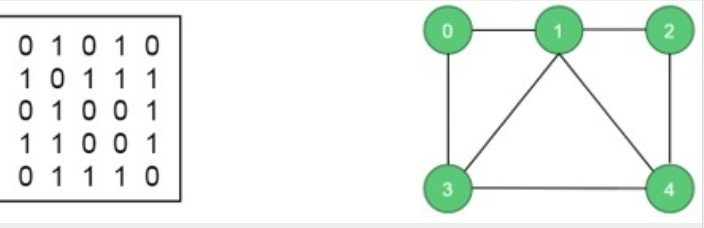


Рисунок 1 – Пример графа, заданного матрицей смежности

При представлении графа матрицей смежности информация о ребрах графа хранится в квадратной матрице, где присутствие пути из одной вершины в другую обозначается весом ребра, иначе нулем.

**Определение:** Цикл, который использует каждую вершину в графе ровно один раз, называется **циклом Гамильтона** , а путь, который использует каждую вершину в графе ровно один раз, называется **гамильтоновым путем.** К сожалению, эта задача намного сложнее, чем соответствующие задачи схемы Эйлера и блуждания; нет хорошей характеризации графов с путями и циклами Гамильтона. Обратите внимание: если у графа есть цикл Гамильтона, то он также имеет путь Гамильтона.

Есть некоторые полезные условия, которые подразумевают существование цикла или пути Гамильтона, которые обычно в той или иной форме говорят о том, что в графе много ребер. Крайним примером является полный граф Kn: у него столько ребер, сколько может иметь любой простой граф с n вершинами, и много циклов Гамильтона. Проблема характеризации состоит в том, что существуют графы с циклами Гамильтона, у которых не так много ребер. Самым простым является цикл Cn: у него всего n ребер, но есть цикл Гамильтона. С другой стороны, на (рисунке 2) показаны графы, у которых всего на несколько ребер больше, чем у цикла на том же числе вершин, но без циклов Гамильтона.

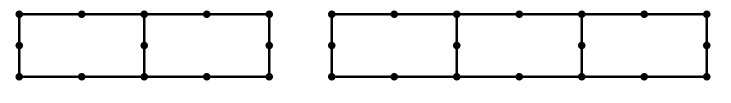


Рисунок 2 – Граф 1 с гамильтоновым путем, но не с гамильтоновым циклом. Граф 2 без цикла и пути

Существуют также графы, у которых, кажется, много ребер, но нет цикла Гамильтона, как показано на (рисунке 3.)

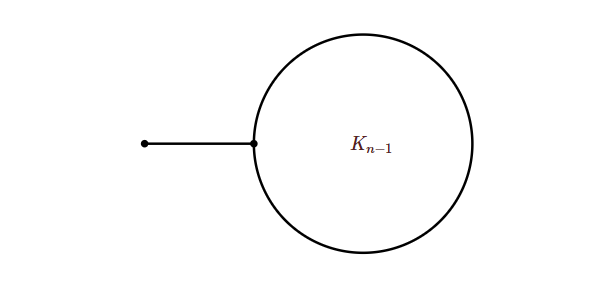


Рисунок 3 – Граф со многими ребрами, но без цикла Гамильтона: полный граф Kn − 1, соединенный ребром с одной вершиной. У этого графа ((n – 1)/2) +1 ребер

Ключ к успешному условию, достаточному для гарантии существования цикла Гамильтона, состоит в том, чтобы потребовать много ребер с большим количеством вершин.

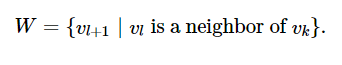
**Теорема. (Ore).**

Если G - простой граф с n вершинами, n≥3 и d (v) + d (w) ≥n, если v и w не смежны, то G имеет цикл Гамильтона.

**Доказательство.**

Сначала покажем, что группа G связна. Если нет, пусть v и w - вершины в двух различных компонентах связности G, и предположим, что компоненты имеют n1 и n2 вершин. Тогда d (v) ≤n1−1 и d (w) ≤n2−1, поэтому d (v) + d (w) ≤n1 + n2−2 <n. Но поскольку v и w не смежны, противоречие.

Теперь рассмотрим максимально длинный путь в G: v1, v2,…, vk. Предположим от противного, что k <n, значит, существует некоторая вершина w, смежная с одной из v2, v3,…, vk − 1, скажем, с vi. Если v1 смежна с vk, то w, vi, vi + 1,…, vk, v1, v2, vi − 1 - путь длины k + 1; противоречие. Следовательно, v1 не смежно с vk, поэтому d (v1) + d (vk) ≥n. Соседи v1 находятся среди {v2, v3,…, vk − 1}, как и соседи vk. Рассмотрим вершины



Тогда , поэтому , множество с k − 1 <n элементами. Поскольку  и W должны иметь общий элемент vj; заметим, что 3≤j≤k − 1. Тогда это цикл длины k:



Для удобства мы можем переименовать вершины:



Теперь, как и раньше, w смежна с некоторым wl, и w, wl, wl + 1,…, wk, w1, w2,…

wl − 1 - путь длины k + 1; противоречие. Таким образом, k = n, и, перенумеровав для удобства вершины, мы получаем гамильтоновский путь v1, v2,…, vn. Если v1 смежна с vn, значит, есть цикл Гамильтона.

Если v1 не смежна с vn, соседи v1 находятся среди {v2, v3,…, vn − 1}, как и соседи vn. Рассмотрим вершины



Тогда , так что , a набор с n − 1 <n элементами.

Поскольку  и W должны иметь общий элемент vi; заметим, что 3≤i≤n − 1. Тогда это цикл длины n:



и является циклом Гамильтона.

# 3. Описание алгоритма поставленной задачи

Пусть G – произвольный граф с n вершинами. Опишем алгоритм с  
возвратом (полного перебора), позволяющий найти в графе G все Гамильтоновы циклы или определить отсутствие таких циклов. Пусть *v*0 – произвольная вершина графа G. Рассмотрим некоторый Гамильтонов цикл  
*v*0 = *u*1, *u*2, *…, u*n, *u*n+1 = *v*0. Если существует алгоритм построения всех простых максимальных цепей с началом в вершине *v*0, то алгоритм нахождении Гамильтоновых циклов также будет существовать. Действительно, пусть

*P*: *v*0 = *u*1, *u*2, *…, u*k

– простая максимальная цепь с началом в вершине *v*0. Если k=n и  
вершина uk смежная с вершиной *v*0, то, добавляя к цепи P ребро  
(*u*k, *v*0), получим Гамильтонов контур.

Алгоритм, позволяющий перечислить по одному разу все Гамильтоновы циклы графа G, многократно выполняет следующие действия: имея  
текущую простую цепь *P*: *v*0 = *u*1, *u*2, *…, u*k-1, он по очереди добавляет к  
ней новые вершины, продолжая ее до простых всевозможных максимальных цепей с началом в вершине *v*0.

Данная программа является многомодульной, поскольку состоит из нескольких функций:

Ниже представлен пвсевдокод функции **void Hamilton();**

1. Инициализируем переменные i, j, k1, k2, k3, k4, n, q, \* adr;

2. Присваиваем в i = Vertex[k - 1](номер последней вершины);

3. Для j = 0; j < cnt\_vertex(кол-во вершин); делать j = j + 1

4. Если элемент исходной матрицы по i-j строке-столбцу не равен 0 [Если (mat[i][j] != 0)]

5. Если (k(номер шага) равен cnt\_vertex(кол-во вершин) и j == 0)

6. Для n = 0; n < cnt\_loop(кол-во Г.циклов); делать n=n+1

7. Находим adr(адрес) = simple\_loop[n]

8. Для k1= 0 Vertex[0] != adr[k1](адресу от k1); делать k1=k1+1

9. Бесконечный цикл , пока выполняется условие

10. Для k3 = k1, k2 = 0; k2 < cnt\_vertex; делать k2=k2+1

11. Если Vertex[k2]!= adr[k3] (вершина от k2 не равна адресу вершины k3)

12. break;

13. Если (k3 == 0)

14. k3 = cnt\_vertex - 1;

15. Иначе

16. --k3

17. Если(k2 == cnt\_vertex)

18. break;

19. Для k3 = k1, k2 = 0, k4 = 0; k2 < cnt\_vertex; делать k2=k2+1

20. Если (Vertex[k4] != adr[k3++ % cnt\_vertex])

21. break;

22. Если (k4 == 0)

23. k4 = cnt\_vertex - 1;

24. else

25. --k4;

26. Если (k2 == cnt\_vertex)

27. break;

28. Если (n == cnt\_loop)

29. Если (cnt\_loop == MAX\_LOOP(максимальное значение))

30. Если (first)

31. Вывести("Количество циклов превысило %d\n", MAX\_LOOP(максимальное значение));

32. first = false;

33. return;

34. simple\_loop[cnt\_loop] = new int[cnt\_vertex];

35. Для (q = 0; q < cnt\_vertex; ++q)

36. simple\_loop[cnt\_loop][q] = Vertex[q];

37. ++cnt\_loop;

38. Иначе если (Mark[j] == 0)

39. Заносим в массив Г. Циклов Vertex[k] = j;

40. Заносим в матрицу посещений 1 Mark[j] = 1;

41. Рекурсивно возвращаемся назад Hamilton(k + 1);

42. Mark[j] = 0;

# 4. Ручной расчёт

Проведем проверку программы посредством ручных вычислений на примере взвешенного неориентированного графа с 5 вершинами. Для наглядности была создана модель.

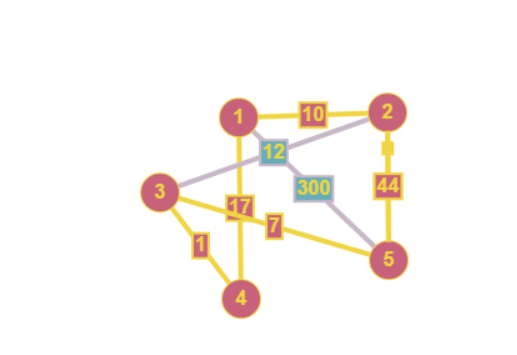


Рисунок 4 – Пример заданного графа

За старшую вершину возьмем 1. Найдем все пути из 1 вершины, проходящий через все другие вершины один раз и возвращающиеся в 1.

1)1-4-3-5-2-1

2)1-2-5-3-4-1 (реверсный 1, длины равны)

3)1-5-2-3-4-1

4)1-4-3-2-5-1 (реверсный 1, длины равны)

Считаем кратчайший Гамильтонов цикл.

1. 17+1+7+44+10 = 79 = 2) 1-2-5-3-4-1
2. 300+44+12+1+17=374 = 4) 1-4-3-2-5-1

Сравниваем длину первого Гамильтонова цикла с третьим найденным нами. Явно видно, что 79 < 374. Получается, что минимальный Гамильтонов цикл равен первому и длина его равна 79. Запишем в файл нашу матрицу, найдем через программу Гамильтоновы циклы и минимальный среди них. Программа работает корректно.

# 5. Описание программы

Для написания данной программы использованы языки программирования С и C++. C - универсальный язык программирования, который завоевал особую популярность у программистов, благодаря сочетанию возможностей языков программирования высокого и низкого уровней. Синтаксис C++ унаследован от языка C. Одним из принципов разработки было сохранение совместимости с C.

Проект был создан в виде консольного приложения Win32 (Visual C++).

Работа программы начинается с основного меню, в котором нужно выбрать, каким образом мы хотим заполнить граф, написав это ключевое слово. Пользователю даётся на выбор 8 действий. Для выбора действия пользователю нужно написать соответствующее слово на клавиатуре и нажать Enter.

// Контроль записи числа

bool test\_num(const char \*s);

Функция тестирует корректность строки, заданной параметром *s*. Строка не должна содержать крайних “пустых” символов (пробелов, табуляций и переводов строк). Возвращается *true*, если строка есть правильное десятичное число; в противном случае возвращается *false*. Функция используется для контроля числовых параметров команд оператора.

// Печать приглашения на ввод команды оператора,

// ввод команды и разбиение ее на параметры

// return false, если введенная строка пустая

// или число параметров слишком велико

// (в этом случае печатается сообщение об ошибке)

bool input\_cmd(void);

Функция формирует массив указателей на параметры команды. Параметры команды отделяются друг от друга произвольным числом “пустых” символов.

Если число параметров равно нулю или больше трех – возвращается значение false; иначе устанавливается в *cnt* количество параметров, формируется массив указателей в *param*. Каждый элемент массива *param* указывает на первый символ соответствующего параметра, в конце каждого параметра содержится двоичный нуль.

// Разбор команды оператора

bool select\_cmd(void);

Функция по коду команды анализирует количество параметров в команде оператора и возвращает *false*, в случае их ошибочного количества. Например, команда *Rand* может содержать два параметра или ни одного.

// Ручной ввод графа

void manual\_graph(void);

Функция выполняет ввод описания графа с консоли. В отличие от ввода графа из файла (там граф определяется матрицей весов), здесь граф вводится в виде ребер с указанием их вершин и веса. Предварительно запрашивается количество ребер и вершин.

// Ввод графа из файла

void load\_graph(void);

Функция по заданному параметром имени файла выполняет ввод матрицы весов графа. Перед вводом матрицы весов вводится количество вершин графа.

// Вернуть true, если граф не задан

bool empty\_graph(void);

Функция используется для контроля наличия введенного графа перед выполнением некоторых команд, например, команды *Matrix* для печати матрицы весов. В случае отсутствия графа печатается сообщение об этом и возвращается *true*.

// Обход графа в глубину для присваивания всем вершинам

// номера области связности

void dfs\_connect(int v);

Функция используется для определения всех вершин, которые связаны ребрами с вершиной *v*. Определение количества областей связности графа необходимо в данной работе, поскольку циклы Гамильтона существуют только для односвязных графов. Для работы функция использует массив *connect*, где для каждой вершины формируется номер области связности, в которую входит соответствующая вершина. Переменная *cnt\_conn*ect содержит количество областей связности.

// Определение компонент связности

void connected\_parts(void);

Функция используется для определения всех областей связности при помощи функции *dfs\_connect*. Каждая не просмотренная вершина графа является параметром функции *dfs\_connect*, которая для этой вершины определяет область связности и все вершины области связности отмечает как просмотренные.

/ Вернуть количество ребер

int get\_cnt\_edge(void);

Возвращает количество ребер графа; используется для контроля возможности построения цикла Гамильтона, поскольку для такого цикла требуется минимум 3 ребра.

// Вернуть true, если граф содержит достаточно ребер и односвязный

bool yes\_loop(FILE \*f);

Функция контролирует граф на наличие обязательных свойств для построения цикла Гамильтона (односвязность и более трех ребер). Результат контроля печатается на консоль или в файл, заданный параметром *f*. Если обязательные свойства соблюдены, возвращается *true*.

// Удалить циклы Гамильтона

void del\_loop(void);

Функция удаляет динамическую память, которая была использована при построении циклов Гамильтона.

// Поиск Гамильтоновых циклов

// k - номер шага

void Hamilton(int k);

Функция получает на входе позицию последней вершины гамильтоновой цепи (обычно берется первая вершина, потому что все вершины графа обязаны быть в гамильтоновом цикле и выбор вершины совершенно произволен). При работе массив Vertex содержит вершины гамильтоновой цепи, которая наращивается еще не просмотренными вершинами графа для превращения цепи в цикл. Так как при этом могут получаться совпадающие прямые или реверсные циклы, выполняется проверка на наличие дубликатов. Все сформированные циклы Гамильтона сохраняются в массиве циклов *simple\_loop*.

Все необходимые функции для поиска циклов Гамильтона используются в void Hamilton

void Hamilton()

{

int i, v;

int check = 0;

int q = 0;

int e=cnt\_vertex/2;

Изначально проверяется существование Гамильтоновых циклов по условию Дирака, для этого мы сравниваем, если степень каждой вершины не меньше, чем cnt\_vertex/2, то данный граф Гамильтонов.

for (int i = 0; i < cnt\_vertex; i++)

{

q = 0;

for (int j = 0; j < cnt\_vertex; j++)

{

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

q++;

}

if (q < e) {

printf(" Граф не содержит Гамильтоновых циклов по Условию Дирака\n");

return;

}

}

Дальше проверяем граф на наличие обязательных свойств для построения цикла Гамильтона (односвязность и более трех ребер). С помощью условия функции yes\_loop

if (!yes\_loop(stdout))

return;

Также дальше мы проверяем условия задания вершины, чтобы не выходила из диапазона.

if (!test\_num(param[1]) || (v = atoi(param[1])) < 1 || v > cnt\_vertex)

{

printf("Вершина ошибочна или вне диапазона [1...%d]\n", cnt\_vertex);

return;

}

Происходит инициализация кол-ва Гамильтоновых циклов cnt\_loop, заполняется массив посещения всех вершин 0, mark[i]=0 , кроме первого элемента mark[0]=1

first = true;

cnt\_loop = 0;

for (i = 0; i < cnt\_vertex; ++i)

Mark[i] = 0;

Mark[0] = 1;

Происходит вызов функции void Hamilton(int k) для поиска циклов , с k=1

Hamilton(1);

Проверка кол-ва Гамильтоновых циклов , т.е. cnt\_loop.

if (cnt\_loop == 0)

Если равно 0 , то выводится

printf("Граф не содержит Гамильтоновых циклов\n");

else

{

Если Гамильтоновы циклы существуют, выводим их на экран и после очищаем память

print\_hamilton(stdout, v);

del\_loop();

}

Поиск самих Гамильтоновых циклов

// Поиск Гамильтоновых циклов

// k - номер шага

void Hamilton(int k)

{

Инициализируем нужные для работы переменные, k=1 из условия.

int i, j, k1, k2, k3, k4, n, q, \* adr;

Ищем номер последней вершины из массива вершин Гам. цепи Vertex

// Номер последней вершины

i = Vertex[k - 1];

Осуществляем поиск Гам. циклов

for (j = 0; j < cnt\_vertex; ++j)

{

Проверяем вершины на наличие ребер

// Есть ребро?

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

{

Есть ребро, сравниваем k c кол-вом вершин cnt\_vertex, и j c нулем. Нет ребра j++ и идем по новой.

if (k == cnt\_vertex && j == 0)

{

Если кол-во гамильтоновых циклов cnt\_loop не равно 0 , то проверяем на совпадение всех вершин этого цикла, что он не реверсно дубликатный.

// Может быть этот цикл уже есть в каком-то виде?

for (n = 0; n < cnt\_loop; ++n)

{

// Находим совпадение по одной вершине

adr = simple\_loop[n];

for (k1 = 0; Vertex[0] != adr[k1]; ++k1)

;

// Есть совпадение всех вершин в прямом/обратном направлении?

for (k3 = k1, k2 = 0; k2 < cnt\_vertex; ++k2)

{

if (Vertex[k2] != adr[k3])

break;

if (k3 == 0)

k3 = cnt\_vertex - 1;

else

--k3;

}

if (k2 == cnt\_vertex)

break;

// Есть совпадение всех вершин в обратном/прямом направлении?

for (k3 = k1, k2 = 0, k4 = 0; k2 < cnt\_vertex; ++k2)

{

if (Vertex[k4] != adr[k3++ % cnt\_vertex])

break;

if (k4 == 0)

k4 = cnt\_vertex - 1;

else

--k4;

}

if (k2 == cnt\_vertex)

break;

if (k2 == cnt\_vertex)

break;

}

// Помещаем цикл в результат, если он не реверсно дубликатный

if (n == cnt\_loop)

{

if (cnt\_loop == MAX\_LOOP)

{

if (first)

{

printf("Количество циклов превысило %d\n", MAX\_LOOP);

first = false;

return;

}

}

Если цикл прошел все условия сохраняем его в массив simple\_loop, c cnt\_loop равным кол-ву циклов по столбцам и Vertex[q] записывается в данный столбец.

simple\_loop[cnt\_loop] = new int[cnt\_vertex];

for (q = 0; q < cnt\_vertex; ++q)

simple\_loop[cnt\_loop][q] = Vertex[q];

++cnt\_loop;

}

}

Если цикл не нашелся, т.е. алгоритм не перебрал все вершины, условие не соблюдено и в матрице посещаемости Mark[j] стоит 0, то

else if (Mark[j] == 0)

{

Заносим в массив Гамильтоновых циклов j

Vertex[k] = j;

Заносим в матрицу посещения вершины 1

Mark[j] = 1;

Если поиск цикла был неудачен - возвращаемся рекурсивно назад к тому месту, откуда еще не были исследованы все пути.

Hamilton(k + 1);

Заносим в матрицу посещаемости 0 и идем дальше перебирать оставшиеся вершины

Mark[j] = 0;

}

}

}

}

// Печать Гамильтоновых циклов

// и выбор из них минимального

void print\_hamilton(FILE \*f, int v);

Функция находит циклы для вершины *v* в массиве *simple\_loop* и печатает их в прямом и реверсном направлении на консоль или в файл *f*. Одновременно выполняется поиск минимального цикла Гамильтона для заданной вершины.

// Сохранение графа

void save\_graph(void);

Функция сохраняет в заданном файле описание графа и все возможные циклы Гамильтона для этого графа.

// Печать матрицы смежности

void print\_matrix(void);

Функция печатает матрицу весов графа на консоль.

// Создание случайного графа

void rand\_graph(void);

Функция по заданному числу вершин и ребер формирует случайный граф со случайными весами. Кроме того, эта функция может формировать полный граф, который точно будет иметь очень много циклов Гамильтона. Гарантируется, что не полный созданный граф будет односвязным (но далеко не всегда Гамильтоновым). Предполагается при дальнейшем изменении программы устранять висячие вершины, наличие которых не позволяет графу иметь циклы Гамильтона.

// Головная функция

int main();

Функция вызывает все описанные выше функции (прямо или косвенно) для создания диалога с пользователем, который при помощи описанных в (пункте 5) команд может управлять работой программы.

1. Help
2. Laod(имя файла)
3. Save(имя файла)
4. Manual ( vertex- число вершин, edge- число ребер)
5. Rand(vertex- число вершин, edge- число ребер)
6. Matrix (вывод на экран полученной матрицы)
7. Hamilton ( vertex- номер вершины)
8. Exit ( Закрытие программы)

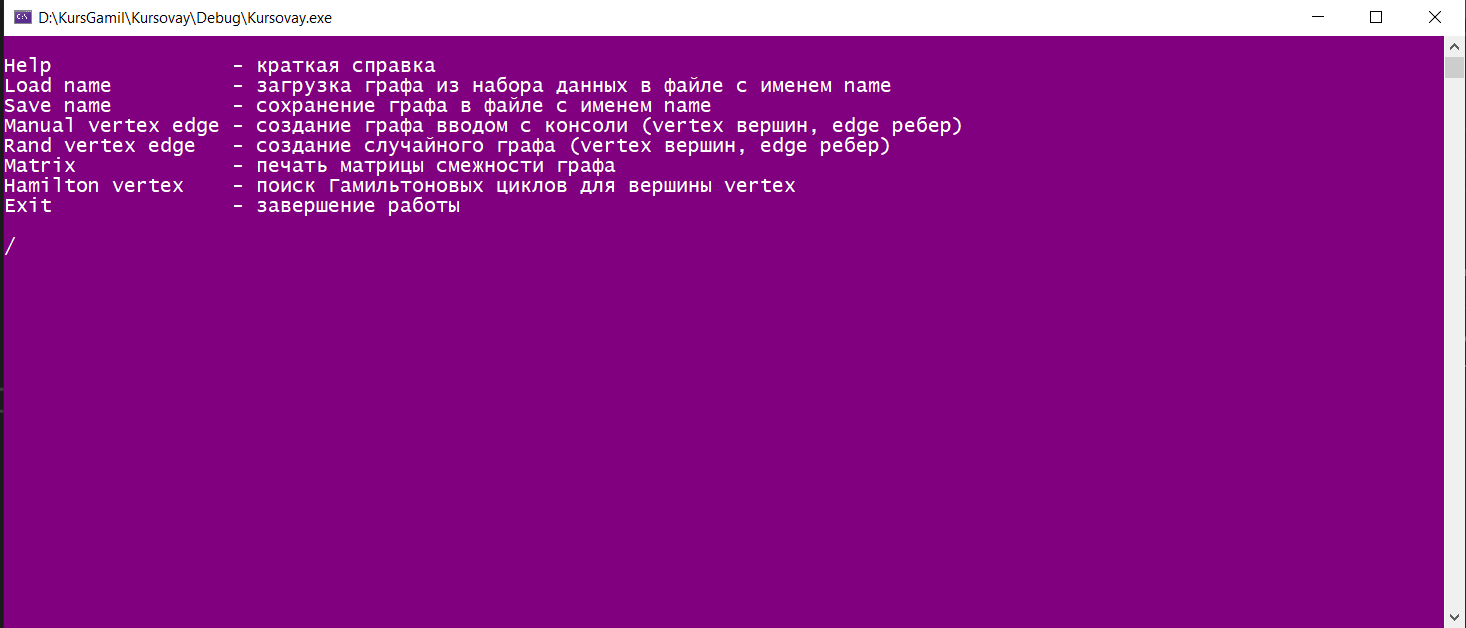
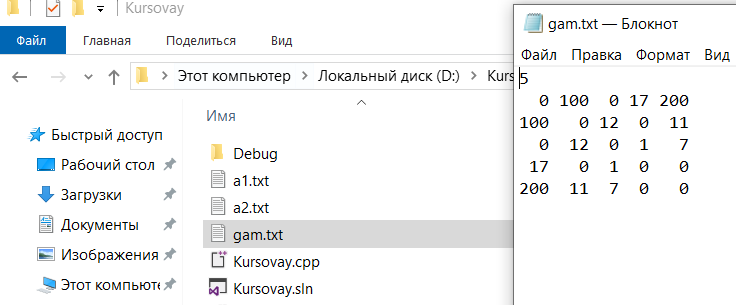


Рисунок 5 – Основное меню

Для заполнения матрицы из файла, мы должны создавать файл формата txt в ключевой папке программы. Файл должен содержать на первой строке кол-во вершин, далее через пробел идут сами вершины. Команда load(название файла) выведет данный файл на экран.



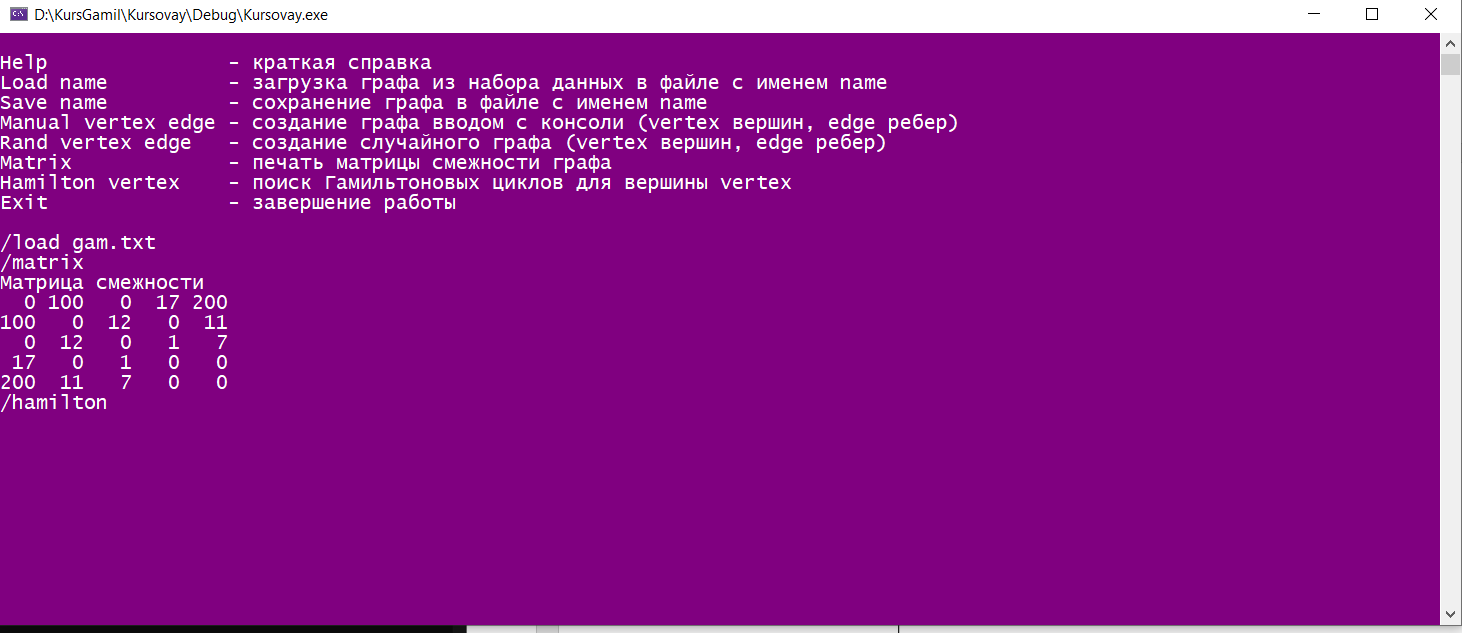


Рисунок 6 – Заполнение матрицы из файла

Для задания графа вручную необходимо ввести с клавиатуры команду manual, задав через пробел количество вершин и ребер в графе. Далее вводится информация о каждом ребре в виде трех чисел: первые два числа – номера вершин ребра (нумерация ведется с 1), а третье число есть расстояние между этими вершинами (стоимость ребра).

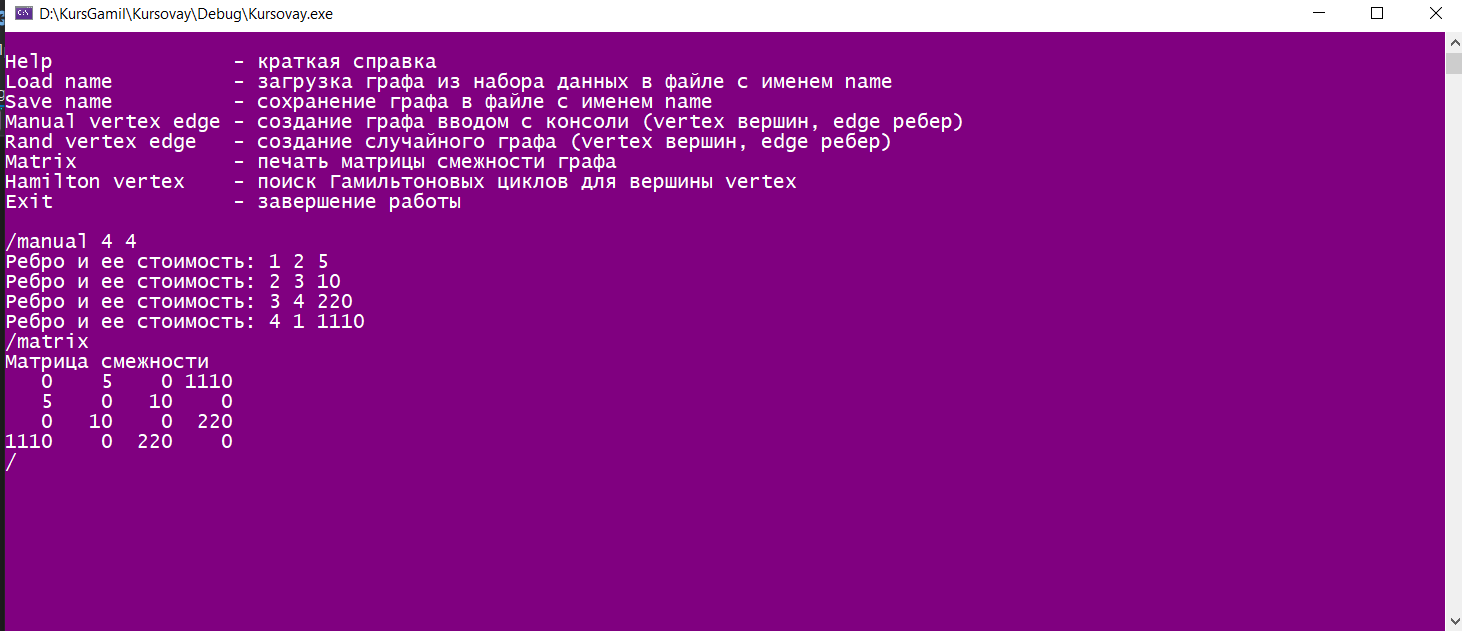


Рисунок 7 – Заполнение матрицы вручную

Для получения случайного графа необходимо ввести с клавиатуры команду rand, задав через пробел количество вершин и ребер в графе. Если rand задается без указания количества ребер, формируется полный граф со случайными стоимостями ребер.

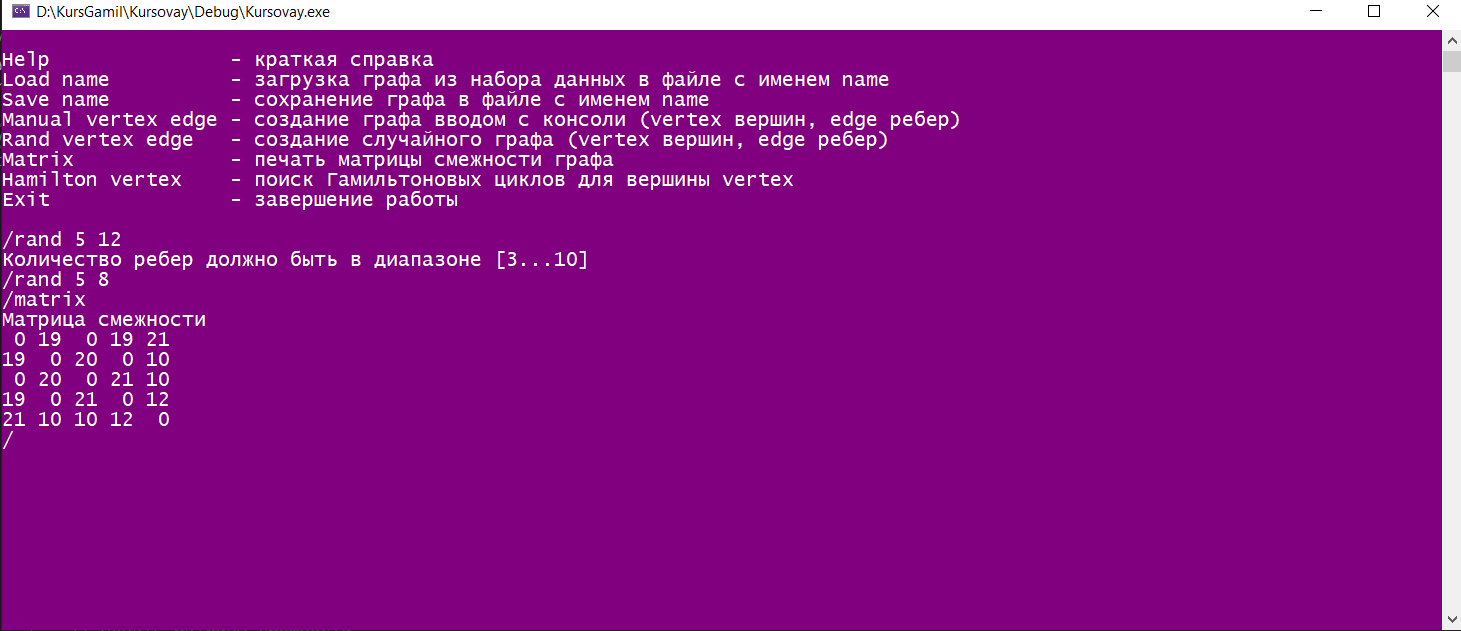


Рисунок 8 – Заполнение матрицы в случайно

Поиск Гамильтоновых циклов выполняется по команде hamilton с номером вершины, для которой строятся все циклы. Кроме того, эта команда находит минимальный цикл среди всех циклов данной вершины.

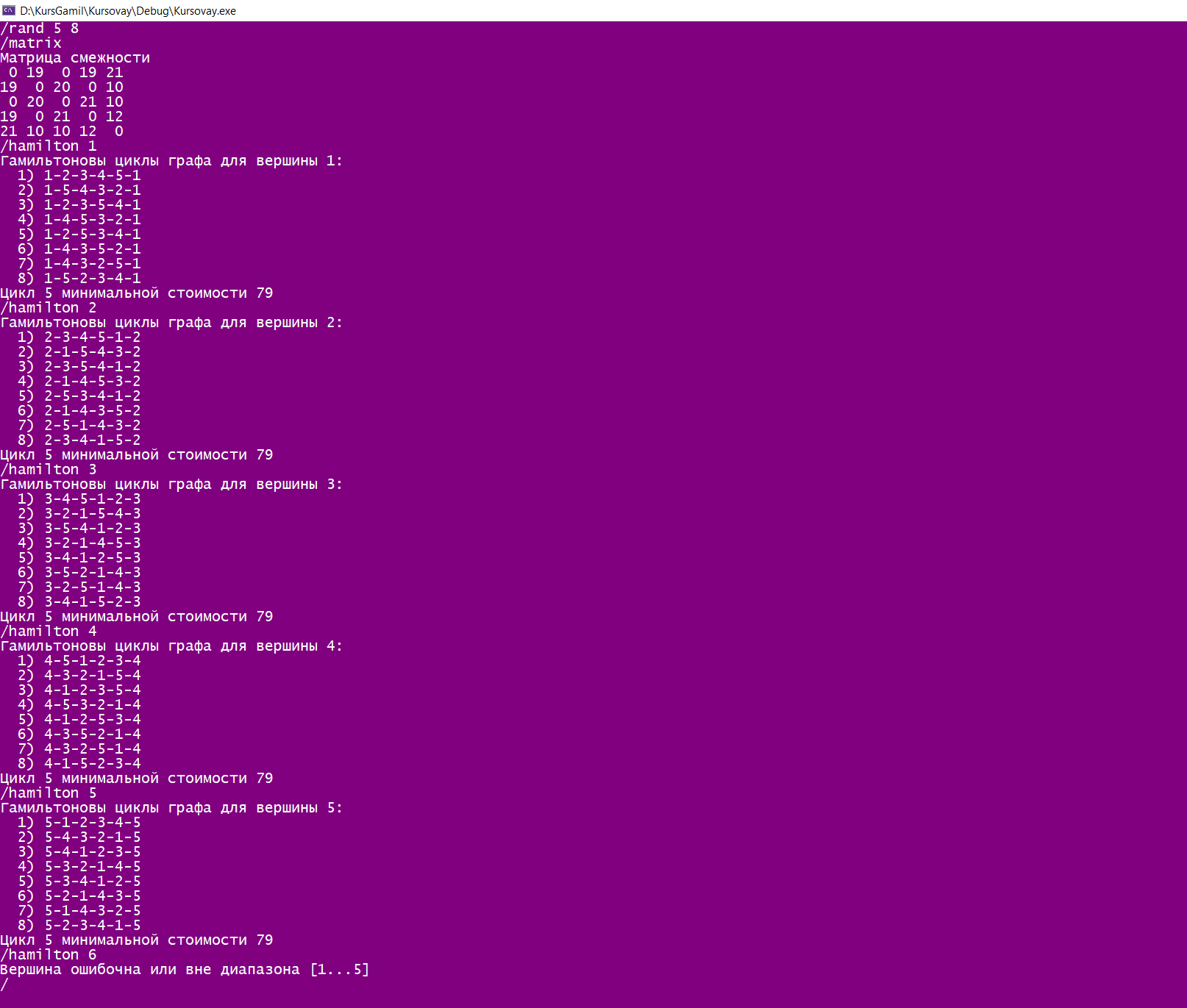


Рисунок 9 – Подсчет гамильтоновых циклов по каждой вершине

Для сохранения в файл наших подсчетов, вводим с клавиатуры команду save (имя файла.txt). Данный файл сохраняется в корневой папке программы.

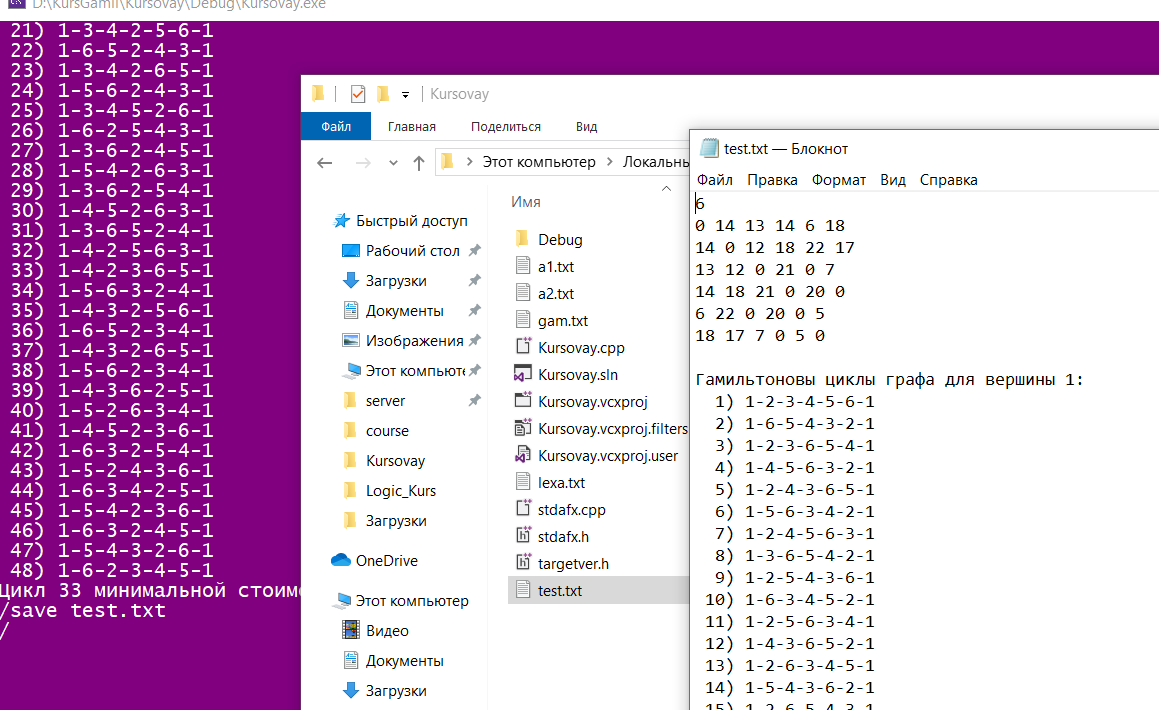


Рисунок 10 – Сохранение в файл

Команда Exit – производит закрытие программы

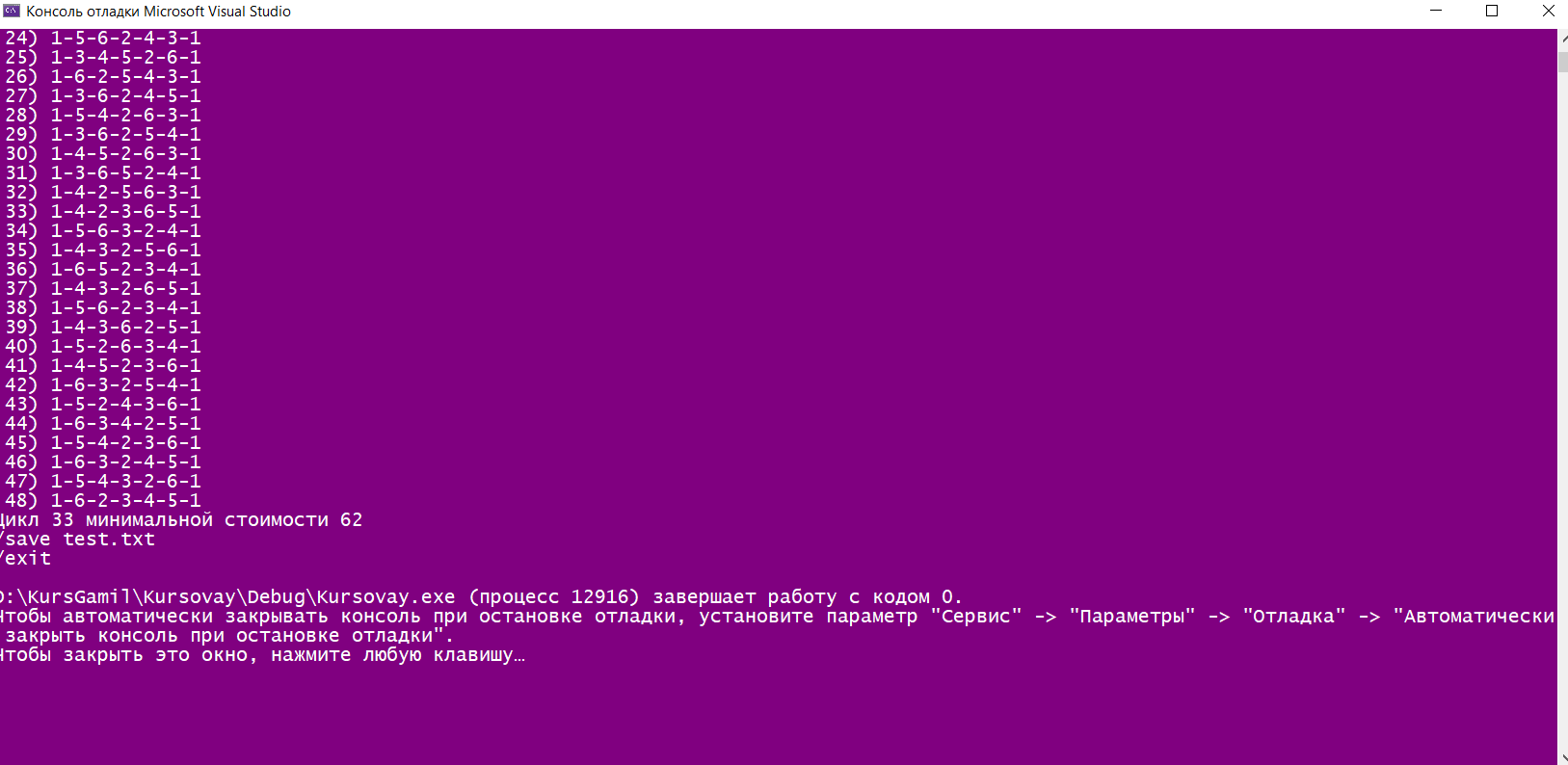


Рисунок 11 – Закрытие программы

Команда help выводит доступные команды, для того , чтобы их вспомнить.

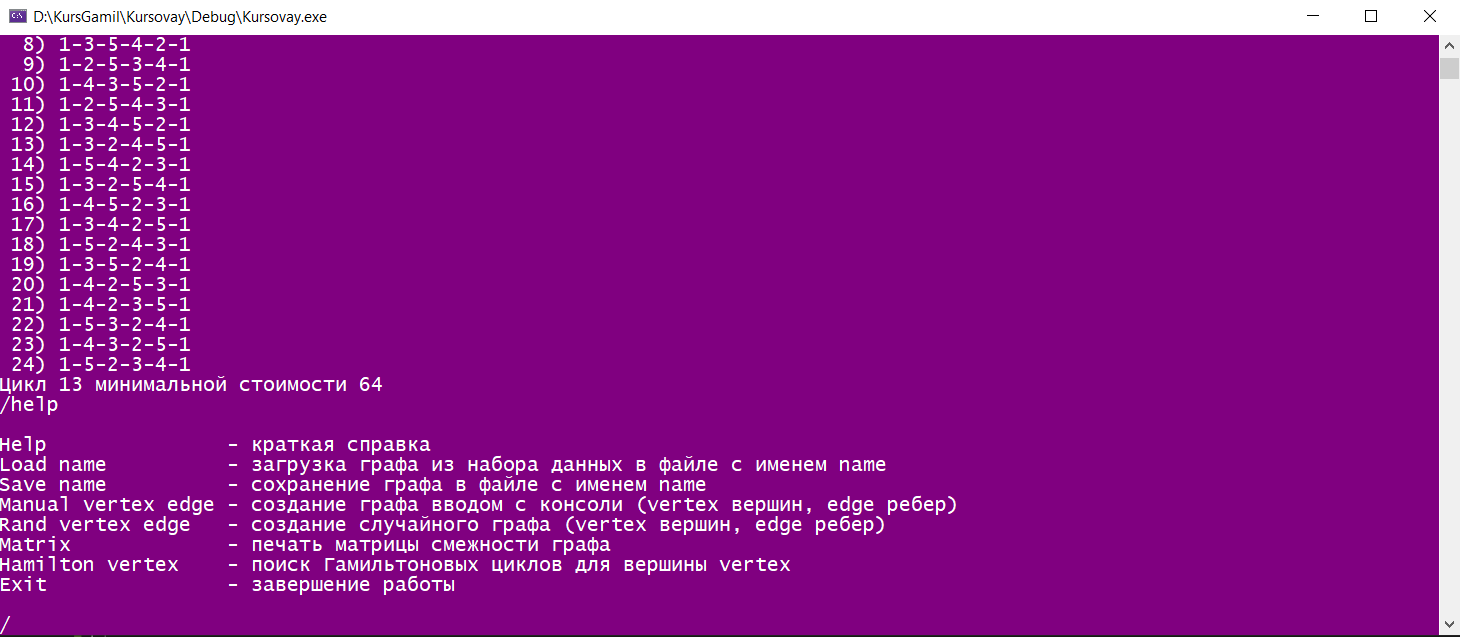


Рисунок 12 – Help-вывод справки на экран

# 6. Тестирование

Среда разработки Microsoft Visual Studio 2019 предоставляет все средства, необходимые при разработке и отладке многомодульной программы.

Тестирование проводилось в рабочем порядке, в процессе разработки, после завершения написания программы. В ходе тестирования было выявлено и исправлено множество проблем, связанных с вводом данных, изменением дизайна выводимых данных, алгоритмом программы, взаимодействием функций.

Для тестирования программы зададим такие данные, которые противоречат нахождению Гамильтоновых циклов. Также проверим программу на потерю данных. Проверим и другие критические моменты.

В таблице ниже продемонстрированы результаты тестирования программы.

Таблица 1 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| Описание теста | Ожидаемый результат | Порядок действий | Полученный результат |
| Ручной ввод | Вывод матрицы с введёнными данными | 1. Ввод команды  manual при запуске программы  2. Последовательный ввод на клавиатуре самого ребра(начала конца) и его веса | Верно |
| Пропускаем одну из компонент ручного ввода | Сообщение о неправильном ручном заполнении. Перенос на след. строку с заполнением | Ввод команды manual с компонентами, кол-вом компонентов не равным 3. | Верно |
| Ввод команды с ошибкой | Вывод ошибки, переход на след. строку для повторного ввода | Ввод команды с ошибкой | Верно |
| Неправильно указанное имя файла | Вывод ошибки о не существовании файла | Ввод названия файла, которого не существует | Верно |
| Выход за границы графа при случайном заполнении | Вывод ошибки , о том , что мы вышли за размер графа. Или ввели число вершин и ребер < 3 , что не соответствует условию. | Ввод количества верши и ребер < 3 или больше заданного нами промежутка | Верно |
| Ищем Гамильтонов цикл из несуществующей вершины | Программа сообщает, что вершина не входит в диапазон | Ввод несуществующей вершины уже заданного графа | Верно |
| Поиск Гамильтоновых циклов в графе, который их не содержит | Программа сообщает нам, что Гамильтоновых циклов не содержит | 1.Задание случайного графа командой rand.  2.Проверка его на Гамильтоновы циклы командой Hamilton. | Верно |
| Случайное заполнение | Генерирование случайной матрицы | Ввод команды rand с  Кол-вом вершин и ребер | Верно |

Тестирование ручного ввода. Программа работает верно!

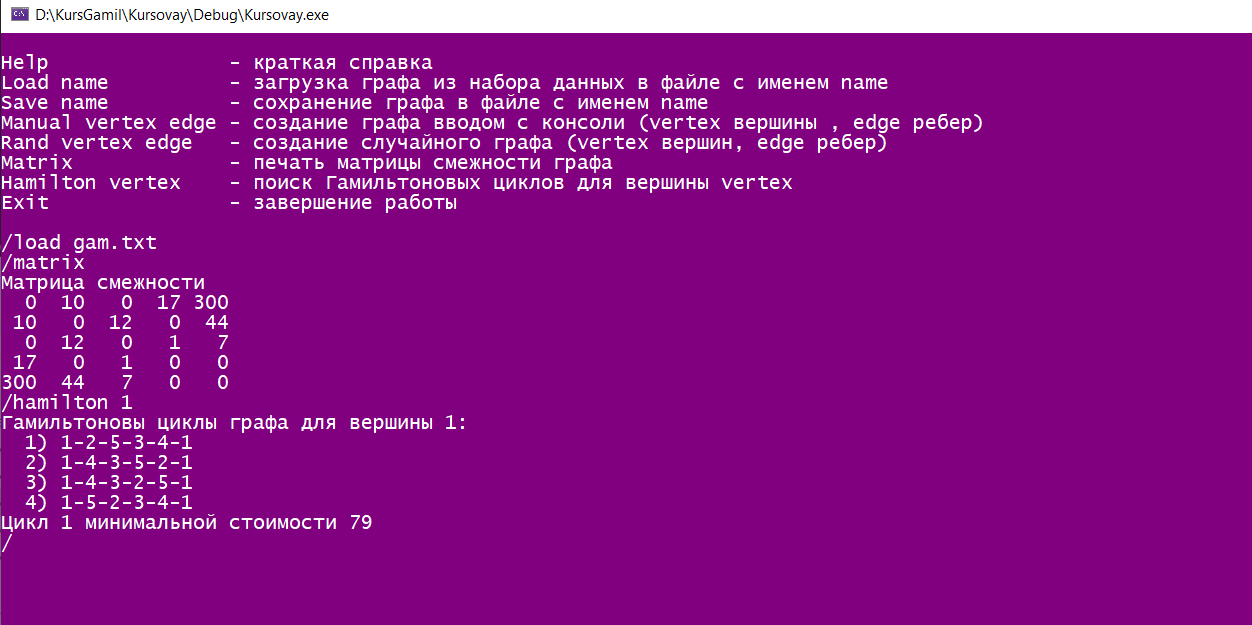


Рисунок 13 – Тест ручного ввода

Тестируем программу при ручном вводе, если мы не ввели какой-либо из 3 компонент, программа пропускает наше действие и предлагает ввести заново. Программа работает верно!

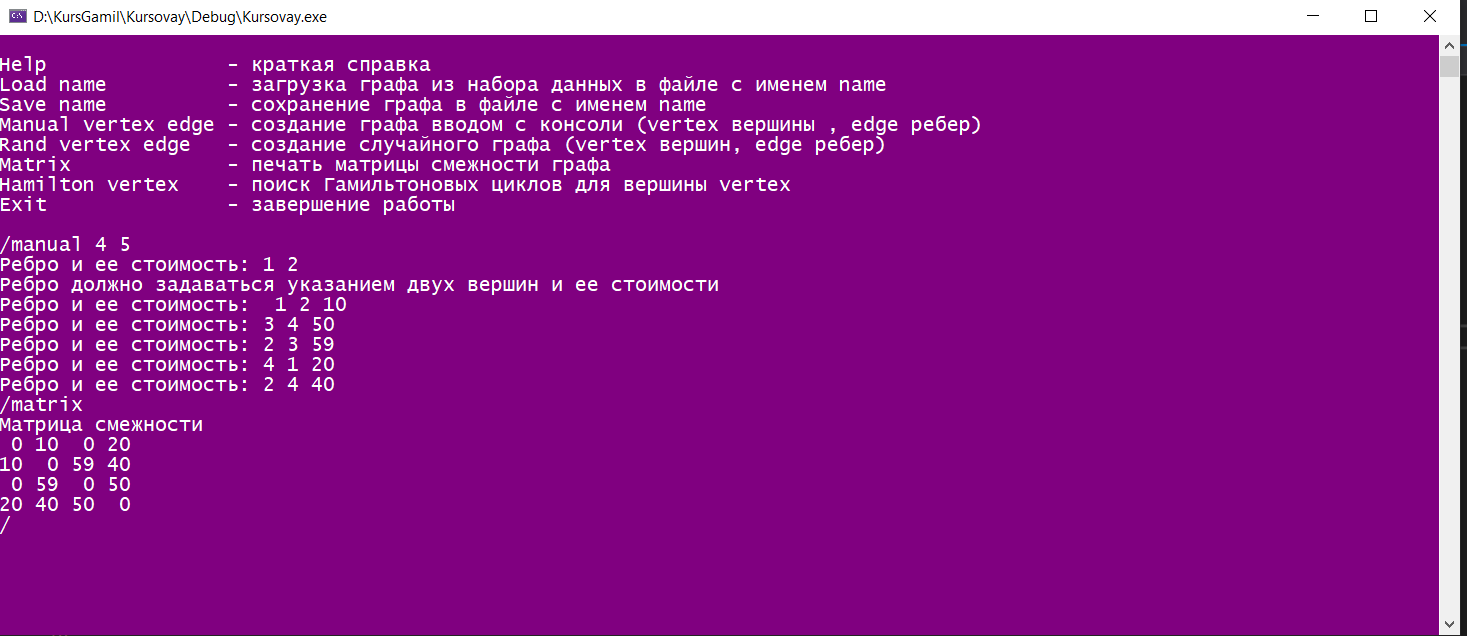


Рисунок 14 – Тест ручного ввода с кол-вом компонентов < > 3

При ошибочном выборе команды, программа оповещает нас об этом и дает возможность ввести команду заново. Программа работает верно!

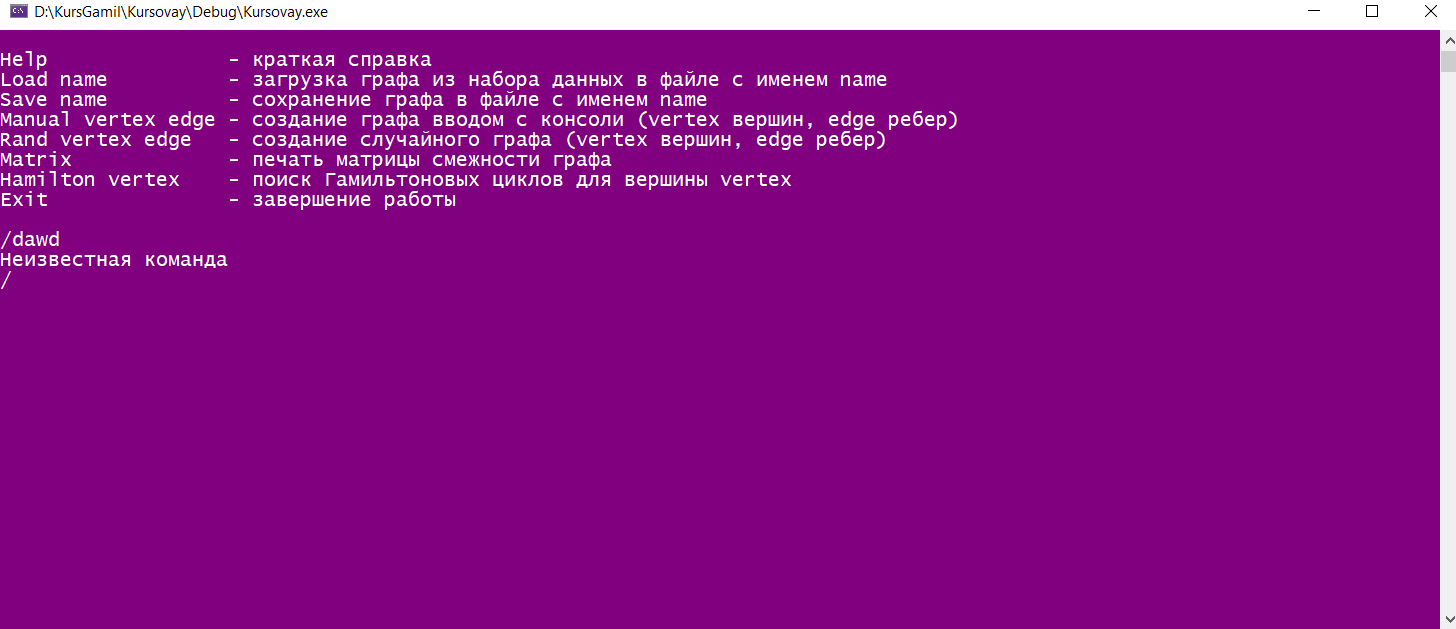


Рисунок 15 – Тест при ошибке в команде

Тестирование при правильно и неправильно указанном файле, для загрузки массива. Программа работает верно!

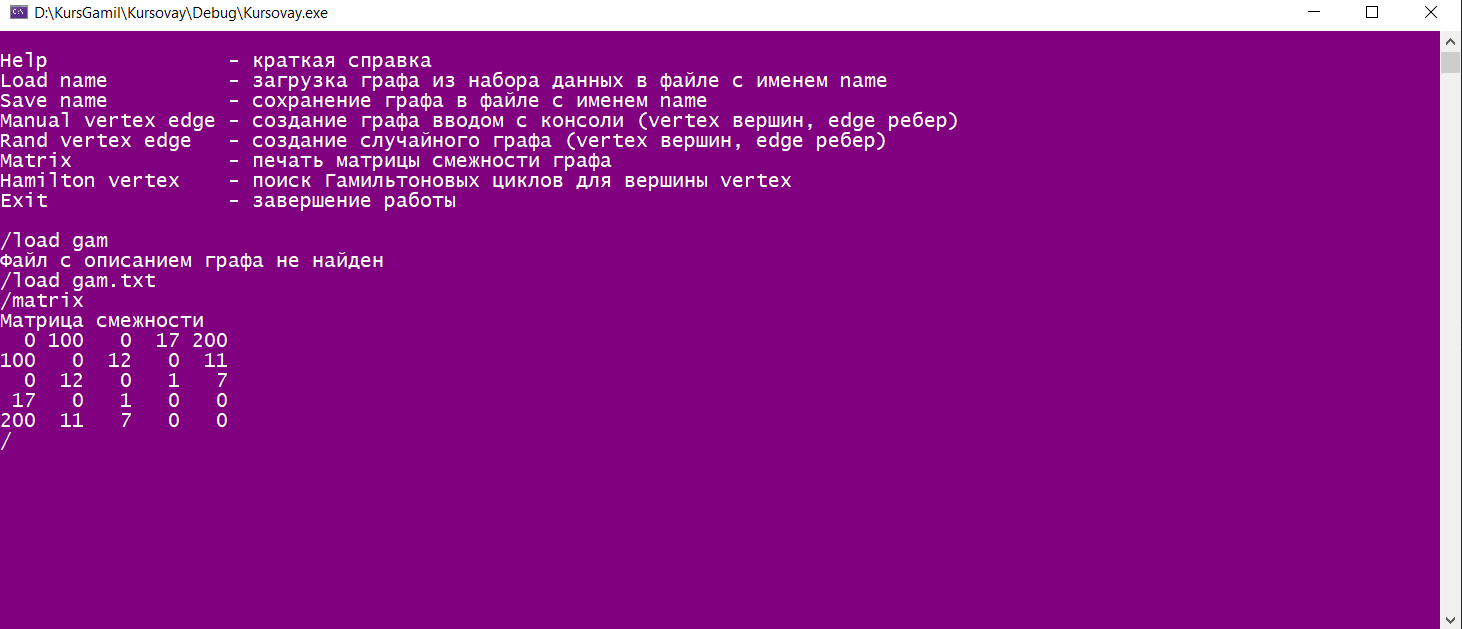


Рисунок 16 – Тест при ошибке в названии файла.

Тестирование при вводе меньше 3 вершин, что не соответствует Гамильтоновым циклам. Также вводим количество ребер, выходящих из диапазона доступных для данного количества вершин. Программа работает верно!

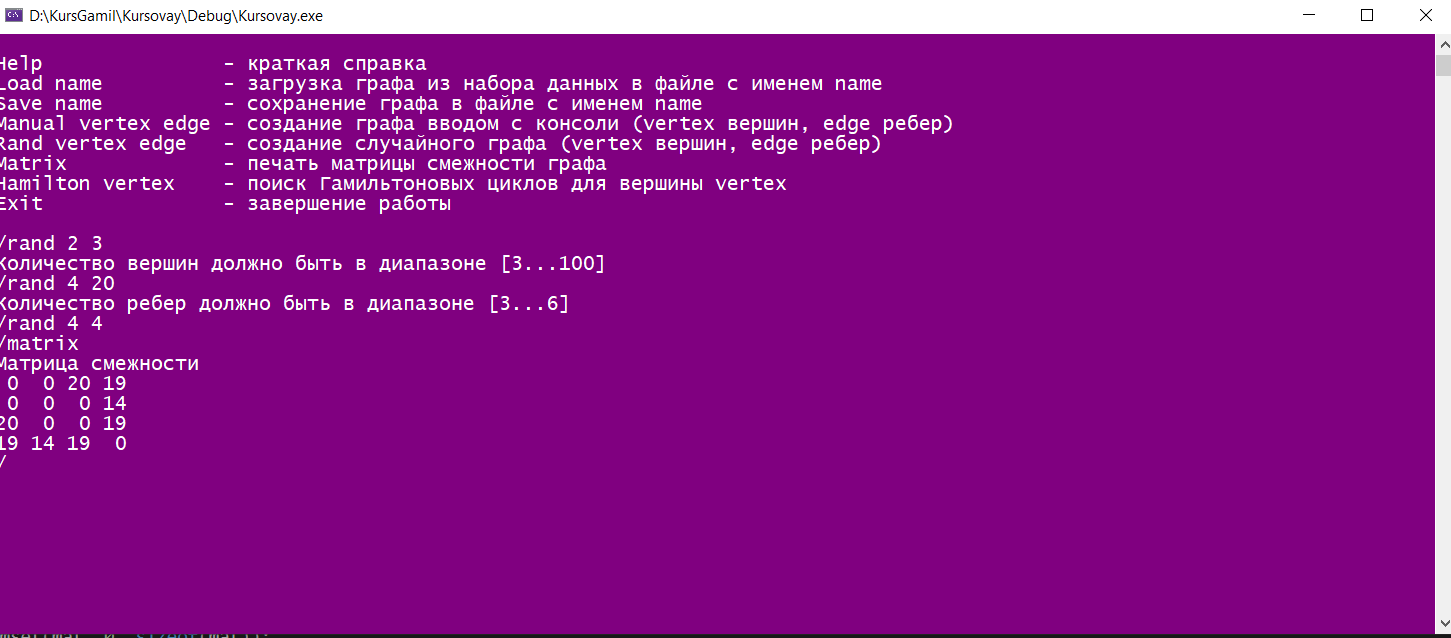


Рисунок 17 – Тест выхода из диапазона ребер и вершин

Тестируем программу на вывод гамильтоновых циклов, из не существующей вершины. Программа работает верно!

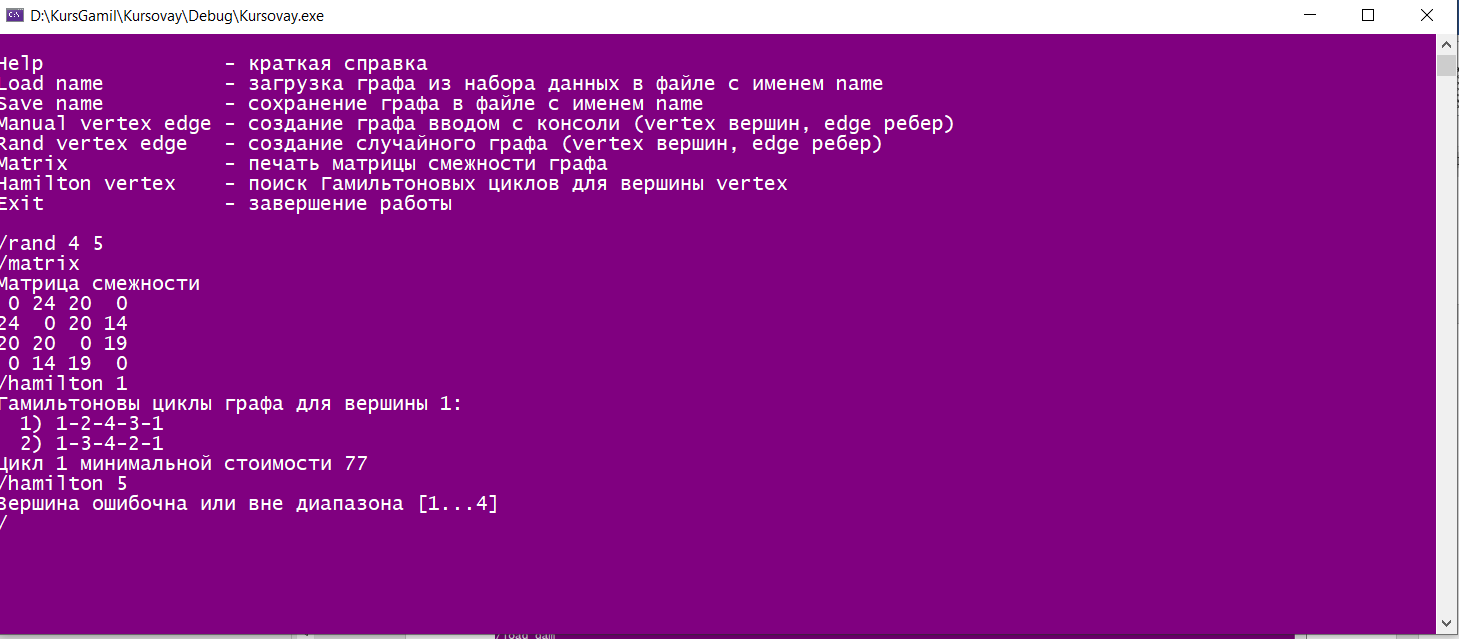


Рисунок 18 – Тест ввода не существующей вершины.

Тестируем программу, создав граф, который не содержит ни одного Гамильтонова цикла. Программа работает верно!

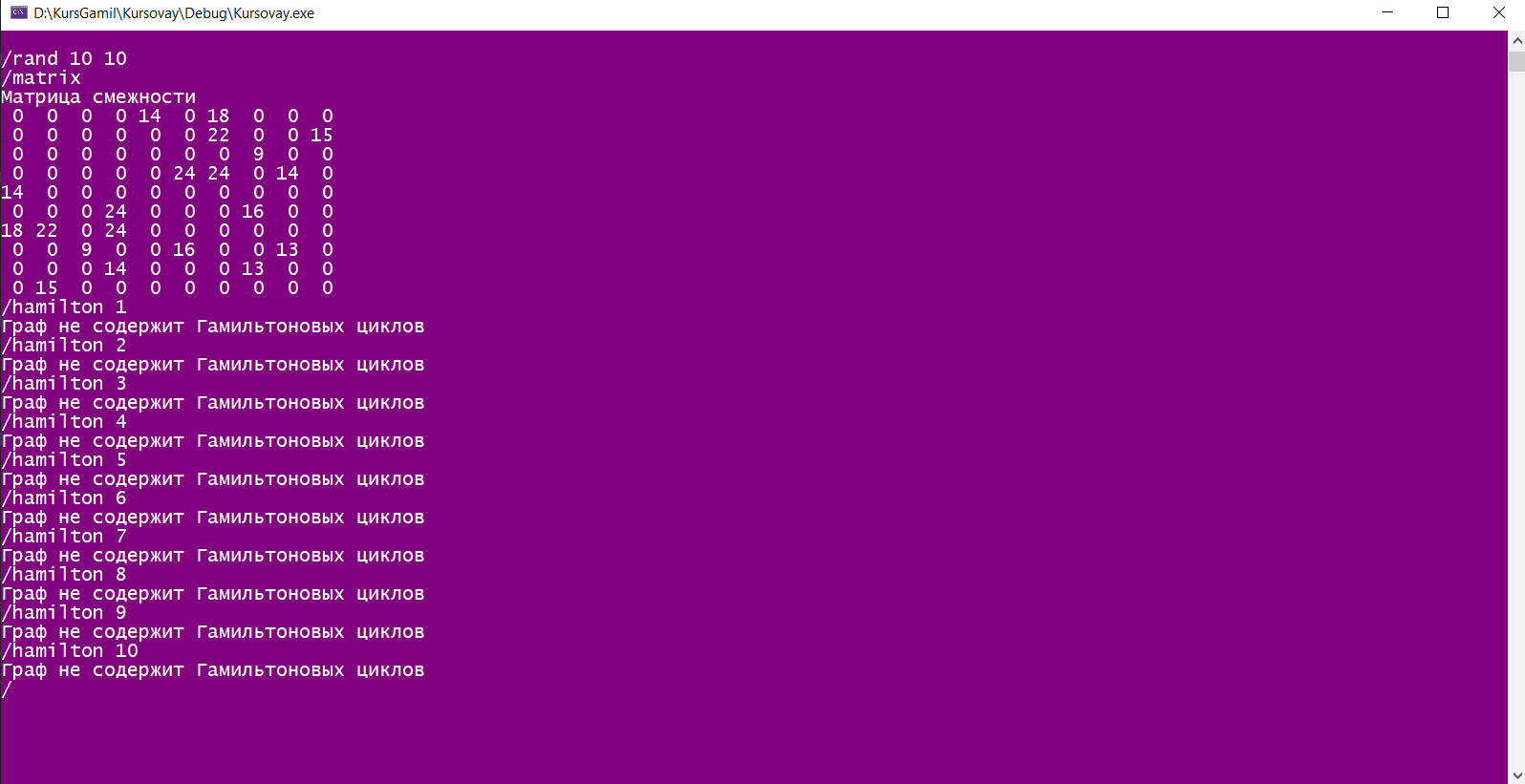


Рисунок 19 – Тест графа без Гам. циклов.

Тестируем программу при вводе всех данных правильно и графа, содержащего Гамильтоновы циклы. Программа работает верно!

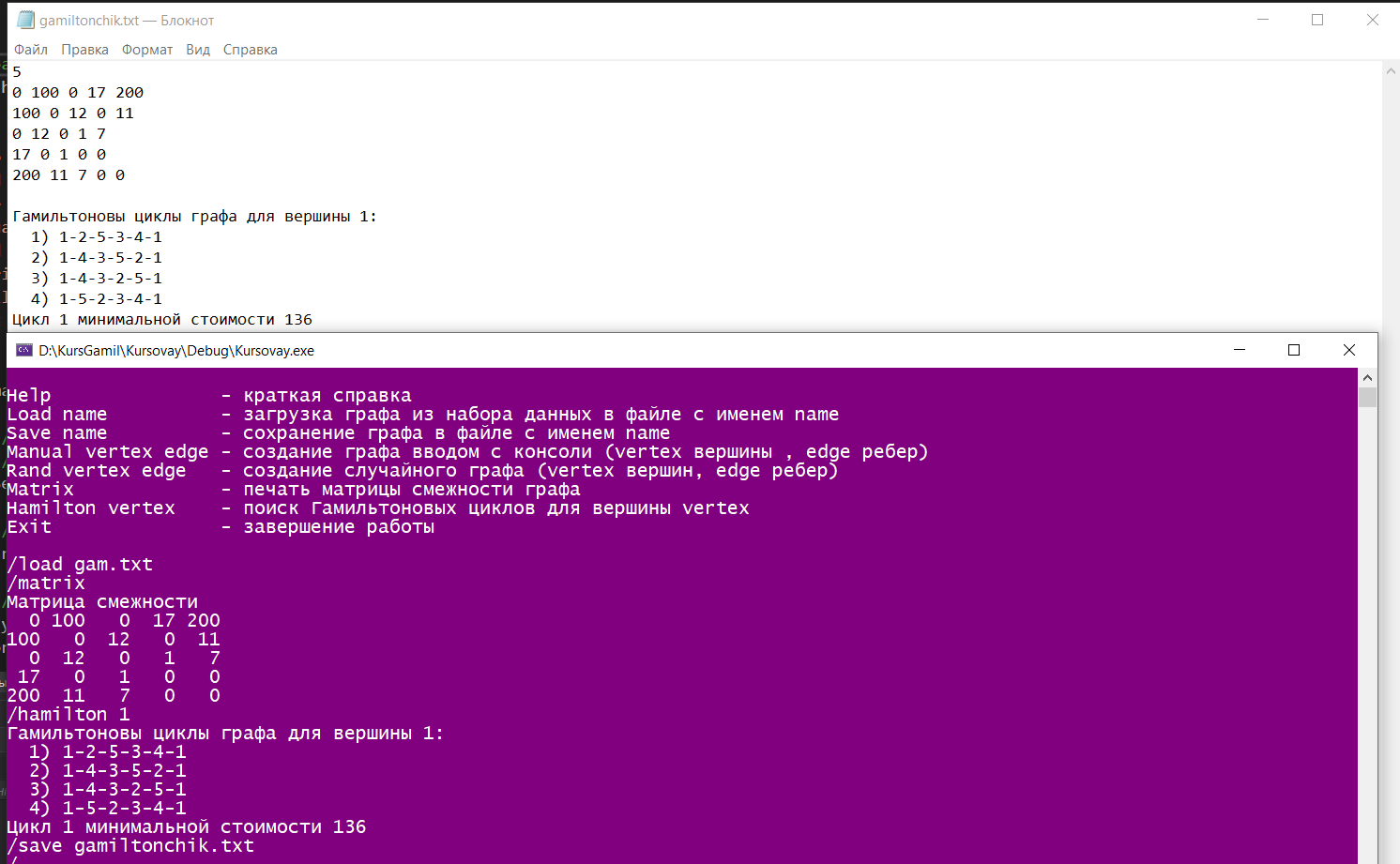


Рисунок 20 – Тест правильного задания графа.

# Заключение

Таким образом, в процессе создания данного проекта разработана программа, которая находит все Гамильтоновы циклы графа, если они в нем есть, а также выводит минимальный по расстоянию среди этих циклов.

При выполнении данной курсовой работы были получены навыки разработки программ и освоены приемы создания матриц смежностей, основанных на теории орграфов. Приобретены навыки по осуществлению алгоритма нахождения Гамильтоновых циклов. Углублены знания языков программирования C и C++.

Недостатком разработанной программы является примитивный пользовательский интерфейс. Потому что программа работает в консольном режиме, не добавляющем к сложности языка сложность программного оконного интерфейса.

Программа имеет небольшой, но достаточный для использования функционал возможностей.

# Список используемых источников

1. Прут В.В. «Графы. Алгоритмы на языке С.» МФТИ 2017.
2. [Зыков А.А. Основы теории графов. - М.:Наука, 1987, 384 с.](https://eek.diary.ru/p178707213.htm#605914263)
3. C/C++. Программирование на языке высокого уровня: Учебник для вузов. Павловская Т. А.
4. Касьянов В. Н., Евстигнеев В. А.,2003. ‘’Графы в программировании: обработка, визуализация и применение’’
5. Теория графов, Омельченко А.В., 2018.

# Приложение

**Листинг программы**

// Hamilton.cpp: определяет точку входа для консольного приложения.

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <SDKDDKVer.h>

#include <stdio.h>

#include <tchar.h>

#include <windows.h>

#include <time.h>

// Максимальное количество вершин в графе

const int MAX\_VERTEX = 100;

// Текущее число вершин

int cnt\_vertex;

int cnt\_vertex2;

// Матрица расстояний неориентированного графа

int mat[MAX\_VERTEX][MAX\_VERTEX];

int mat2[MAX\_VERTEX][MAX\_VERTEX];

// Массив для определения номера области связности для каждой вершины

int connectx[MAX\_VERTEX];

// Количество областей связности

int cnt\_connect;

// Вершины с одной областью связности

int connecty[MAX\_VERTEX];

// Максимальное число циклов

const int MAX\_LOOP = 6000;

// Массив циклов

int\* simple\_loop[MAX\_LOOP];

// Количество циклов

int cnt\_loop;

int Mark[MAX\_VERTEX];

int Vertex[MAX\_VERTEX];

// Коды команд на выполнение

enum

{

HELP,

MATRIX,

HAMILTON,

RAND,

MANUAL,

LOAD,

SAVE,

EXIT

};

// Команды оператора

struct CODE

{

int code;

const char\* s;

};

struct CODE code[] =

{

{ HELP, "help" },

{ MATRIX, "matrix" },

{ HAMILTON, "hamilton" },

{ RAND, "rand" },

{ MANUAL, "manual" },

{ LOAD, "load" },

{ SAVE, "save" },

{ EXIT, "exit" },

{ 0, NULL }

};

// Строка с командой оператора

char buf[1024];

// Код команды

int command;

// Количество параметров в команде

int cnt;

// Параметр в виде числа

int num;

// Максимальное число параметров команды

const int MAX\_P = 3;

// Массив указателей на параметры

const char\* param[MAX\_P];

// Контроль записи числа

bool test\_num(const char\* s)

{

if (!isdigit(\*s))

return false;

while (isdigit(\*++s))

;

return \*s == 0;

}

// Печать приглашения на ввод команды оператора,

// ввод команды и разбиение ее на параметры

// return false, если введенная строка пустая

// или число параметров слишком велико

// (в этом случае печатается сообщение об ошибке)

bool input\_cmd(void)

{

char delim[] = " \t\n", \* txt, \* word;

int i;

const char\* s;

// Приглашение и ввод

printf("/");

fgets(txt = buf, sizeof(buf) - 1, stdin);

// Получаем параметры

for (cnt = 0; (word = strtok(txt, delim)) != NULL; txt = NULL, ++cnt)

{

if (cnt == MAX\_P)

{

printf("Команда содержит слишком много параметров\n");

return false;

}

param[cnt] = word;

}

// Команда не введена?

if (cnt == 0)

return false;

// Распознаем тип команды

for (i = 0; (s = code[i].s) != NULL; ++i)

{

if (\_stricmp(s, param[0]) == 0)

{

command = code[i].code;

return true;

}

}

printf("Неизвестная команда\n");

return false;

}

// Разбор команды оператора

bool select\_cmd(void)

{

// Получить список параметров введенной команды

if (!input\_cmd())

return false;

// Контролируем число параметров

switch (command)

{

case LOAD:

case SAVE:

case HAMILTON:

if (cnt != 2) // Параметр -1 т.е. количество параметров не равно 1

{

printf("Ошибочно число параметров\n");

return false;

}

return true;

case RAND:

if (cnt != 3)

{

printf("Ошибочно число параметров\n");

return false;

}

return true;

case MANUAL:

if (cnt != 3)

{

printf("Ошибочно число параметров\n");

return false;

}

return true;

}

if (cnt != 1)

{

printf("Ошибочно число параметров\n");

return false;

}

return true;

}

// Ручной ввод графа

void manual\_graph(void)

{

int i, n1, n2, v1, v2, v;

char delim[] = " \t\n", \* txt, \* word;

if (!test\_num(param[1]))

{

printf("Количество вершин ошибочно\n");

return;

}

if ((n1 = atoi(param[1])) < 3 || n1 > MAX\_VERTEX)

{

printf("Количество вершин должно быть в диапазоне [3...%d]\n", MAX\_VERTEX);

return;

}

if (!test\_num(param[2]))

{

printf("Количество ребер ошибочно\n");

return;

}

if ((n2 = atoi(param[2])) < 3 || n2 > (n1 \* (n1 - 1)) / 2)

{

printf("Количество ребер должно быть в диапазоне [3...%d]\n", (n1 \* (n1 - 1)) / 2);

return;

}

cnt\_vertex = n1;

memset(mat, 0, sizeof(mat));

// Вводим ребра со стоимостью

for (i = 0; i < n2;)

{

printf("Ребро и ее стоимость: ");

fgets(txt = buf, sizeof(buf) - 1, stdin);

// Получаем параметры

for (cnt = 0; (word = strtok(txt, delim)) != NULL; txt = NULL, ++cnt)

{

if (cnt == 3)

{

++cnt;

break;

}

param[cnt] = word;

}

if (cnt != 3)

{

printf("Ребро должно задаваться указанием двух вершин и ее стоимости\n");

continue;

}

if (!test\_num(param[0]) || (v1 = atoi(param[0])) < 1 || v1 > n1)

{

printf("Первая вершина ошибочна\n");

continue;

}

if (!test\_num(param[1]) || (v2 = atoi(param[1])) < 1 || v1 > n1)

{

printf("Вторая вершина ошибочна\n");

continue;

}

if (v1 == v2)

{

printf("Вершины не должны совпадать\n");

continue;

}

if (!test\_num(param[2]) || (v = atoi(param[2])) < 1)

{

printf("Ошибочна стоимость ребра\n");

continue;

}

// Устанавливаем симметрично вес ребра

--v1;

--v2;

mat[v1][v2] = mat[v2][v1] = v;

++i;

}

}

// Ввод графа из файла

void load\_graph(void)

{

FILE\* f;

int i, j;

bool zero;

if ((f = fopen(param[1], "r")) == NULL)

{

printf("Файл с описанием графа не найден\n");

return;

}

if (fscanf(f, "%d", &cnt\_vertex2) != 1)

{

printf("Ошибочно количество вершин\n");

fclose(f);

return;

}

if (cnt\_vertex2 < 2 || cnt\_vertex2 > MAX\_VERTEX)

{

printf("Количество вершин должно быть в диапазоне [2...%d]\n", MAX\_VERTEX);

fclose(f);

return;

}

// Вводим матрицу расстояний

for (i = 0; i < cnt\_vertex2; ++i)

{

for (j = 0; j < cnt\_vertex2; ++j)

{

if (fscanf(f, "%d", mat2[i] + j) != 1)

{

printf("Ошибочно расстояние A[%d][%d]\n", i, j);

fclose(f);

return;

}

if (mat2[i][j] < 0)

{

printf("Отрицательное расстояние A[%d][%d]\n", i, j);

fclose(f);

return;

}

}

}

fclose(f);

// Окончательный контроль матрицы расстояний

zero = true;

for (i = j = 0; i < cnt\_vertex2; ++i)

{

if (mat2[i][i])

{

printf("Матрица расстояний должна содержать нулевую диагональ, A[%d][%d] != 0\n", i, i);

return;

}

for (j = 0; j < cnt\_vertex2; ++j)

{

if (mat2[i][j])

zero = false;

if (mat2[i][j] != mat2[j][i])

{

printf("Матрица расстояний должна быть симметричной, A[%d][%d] != A[%d][%d]\n", i, j, j, i);

return;

}

}

}

if (zero)

{

printf("Матрица расстояний нулевая\n");

return;

}

// Устанавливаем матрицу расстояний

memcpy(mat, mat2, sizeof(mat2));

cnt\_vertex = cnt\_vertex2;

}

// Вернуть true, если граф не задан

bool empty\_graph(void)

{

if (cnt\_vertex == 0)

{

printf("Граф не задан\n");

return true;

}

return false;

}

// Обход графа в глубину для присваивания всем вершинам

// номера области связности

void dfs\_connect(int v)

{

int i;

// Помечаем текущую вершину как посещенную и принадлежащую cnt\_connect-й компоненте связности

connectx[v] = cnt\_connect;

// Перебираем все вершины смежные с v, которые еще не были посещены

for (i = 0; i < cnt\_vertex; ++i)

{

if (mat[v][i] != 0 && connectx[i] == 0)

{

// Повторяем алгоритм для i-й вершины

dfs\_connect(i);

}

}

}

// Определение компонент связности

void connected\_parts(void)

{

int i;

// Номер очередной компоненты связности

cnt\_connect = 0;

// Сброс номеров компонент связности

memset(connectx, 0, sizeof(connectx));

for (i = 0; i < cnt\_vertex; ++i)

{

// Если вершина еще не была посещена, начинаем

// из нее поиск новой компоненты связности

if (connectx[i] == 0)

{

// Увеличиваем число областей связности

++cnt\_connect;

// Просматриваем все вершины, достижимые из i-й

dfs\_connect(i);

}

}

}

// Вернуть количество ребер

int get\_cnt\_edge(void)

{

int i, j, m;

for (i = m = 0; i < cnt\_vertex; ++i)

{

for (j = 0; j < cnt\_vertex; ++j)

{

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

++m;

}

}

return m;

}

// Вернуть true, если граф содержит достаточно ребер и односвязный

bool yes\_loop(FILE\* f)

{

// А есть ли граф?

if (empty\_graph())

return false;

// Определение компонент связности

connected\_parts();

if (cnt\_connect != 1)

{

fprintf(f, "Граф не односвязный, областей связности = %d\n", cnt\_connect);

return false;

}

cnt\_connect = 0;

// Достаточно ли ребер для построения хотя бы одного цикла?

if (get\_cnt\_edge() / 2 < 3)

{

fprintf(f, "Для цикла необходимо хотя бы 3 ребра\n");

return false;

}

return true;

}

// Удалить циклы Гамильтона

void del\_loop(void)

{

int i;

for (i = 0; i < cnt\_loop; ++i)

delete simple\_loop[i];

}

bool first;

// Поиск Гамильтоновых циклов

// k - номер шага

void Hamilton(int k)

{

int i, j, k1, k2, k3, k4, n, q, \* adr;

// Номер последней вершины

i = Vertex[k - 1];

for (j = 0; j < cnt\_vertex; ++j)

{

// Есть ребро?

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

{

if (k == cnt\_vertex && j == 0)

{

// Может быть этот цикл уже есть в каком-то виде?

for (n = 0; n < cnt\_loop; ++n)

{

// Находим совпадение по одной вершине

adr = simple\_loop[n];

for (k1 = 0; Vertex[0] != adr[k1]; ++k1)

;

// Есть совпадение всех вершин в прямом/прямом направлении?

for (k3 = k1, k2 = 0; k2 < cnt\_vertex; ++k2)

{

if (Vertex[k2] != adr[k3++ % cnt\_vertex])

break;

}

if (k2 == cnt\_vertex)

break;

// Есть совпадение всех вершин в прямом/обратном направлении?

for (k3 = k1, k2 = 0; k2 < cnt\_vertex; ++k2)

{

if (Vertex[k2] != adr[k3])

break;

if (k3 == 0)

k3 = cnt\_vertex - 1;

else

--k3;

}

if (k2 == cnt\_vertex)

break;

// Есть совпадение всех вершин в обратном/прямом направлении?

for (k3 = k1, k2 = 0, k4 = 0; k2 < cnt\_vertex; ++k2)

{

if (Vertex[k4] != adr[k3++ % cnt\_vertex])

break;

if (k4 == 0)

k4 = cnt\_vertex - 1;

else

--k4;

}

if (k2 == cnt\_vertex)

break;

// Есть совпадение всех вершин в обратном/обратном направлении?

for (k3 = k1, k2 = 0, k4 = 0; k2 < cnt\_vertex; ++k2)

{

if (Vertex[k4] != adr[k3])

break;

if (k4 == 0)

k4 = cnt\_vertex - 1;

else

--k4;

if (k3 == 0)

k3 = cnt\_vertex - 1;

else

--k3;

}

if (k2 == cnt\_vertex)

break;

}

// Помещаем цикл в результат, если он не реверсно дубликатный

if (n == cnt\_loop)

{

if (cnt\_loop == MAX\_LOOP)

{

if (first)

{

printf("Количество циклов превысило %d\n", MAX\_LOOP);

first = false;

return;

}

}

simple\_loop[cnt\_loop] = new int[cnt\_vertex];

for (q = 0; q < cnt\_vertex; ++q)

simple\_loop[cnt\_loop][q] = Vertex[q];

++cnt\_loop;

}

}

else if (Mark[j] == 0)

{

Vertex[k] = j;

Mark[j] = 1;

Hamilton(k + 1);

Mark[j] = 0;

}

}

}

}

// Печать Гамильтоновых циклов

// и выбор из них минимального

void print\_hamilton(FILE\* f, int v)

{

int i, i2 = 0, j, k, k2, min = 0, v1, v2, s, ind = 0;

int\* adr;

fprintf(f, "Гамильтоновы циклы графа для вершины %d:\n", v--);

for (i = 0; i < cnt\_loop; ++i)

{

// Находим позицию вершины в Гамильтоне

adr = simple\_loop[i];

for (k = 0; adr[k] != v; ++k)

;

// Печатаем Гамильтон в прямом направлении

// и определяем минимальный Гамильтон

fprintf(f, "%3d) ", ++i2);

fprintf(f, "%d", v + 1);

v1 = v;

for (j = s = 0; j < cnt\_vertex; ++j)

{

fprintf(f, "-%d", (v2 = adr[(k + j + 1) % cnt\_vertex]) + 1);

s += mat[v1][v2];

v1 = v2;

}

fprintf(f, "\n");

if (min == 0 || min > s)

{

min = s;

ind = i2;

}

// Печатаем Гамильтон в обратном направлении

fprintf(f, "%3d) ", ++i2);

fprintf(f, "%d", v + 1);

v1 = v;

k2 = k;

for (j = s = 0; j < cnt\_vertex; ++j)

{

if (--k2 < 0)

k2 = cnt\_vertex - 1;

fprintf(f, "-%d", adr[k2] + 1);

}

fprintf(f, "\n");

}

fprintf(f, "Цикл %d минимальной стоимости %d\n", ind, min);

}

void Hamilton()

{

int i, v;

if (!yes\_loop(stdout))

return;

if (!test\_num(param[1]) || (v = atoi(param[1])) < 1 || v > cnt\_vertex)

{

printf("Вершина ошибочна или вне диапазона [1...%d]\n", cnt\_vertex);

return;

}

first = true;

cnt\_loop = 0;

for (i = 0; i < cnt\_vertex; ++i)

Mark[i] = 0;

Mark[0] = 1;

Hamilton(1);

if (cnt\_loop == 0)

printf("Граф не содержит Гамильтоновых циклов\n");

else

{

print\_hamilton(stdout, v);

del\_loop();

}

}

// Сохранение графа

void save\_graph(void)

{

FILE\* f;

int i, j;

if (empty\_graph())

return;

if ((f = fopen(param[1], "w")) == NULL)

{

printf("Нельзя создать файл\n");

return;

}

// Сохраняем количество вершин

fprintf(f, "%d\n", cnt\_vertex);

// Построчно сохраняем матрицу смежности

for (i = 0; i < cnt\_vertex; ++i)

{

for (j = 0; j < cnt\_vertex; ++j)

{

fprintf(f, "%d ", mat[i][j]);

}

fprintf(f, "\n");

}

fprintf(f, "\n");

if (yes\_loop(f))

{

first = true;

cnt\_loop = 0;

for (i = 0; i < cnt\_vertex; ++i)

Mark[i] = 0;

Mark[0] = 1;

Hamilton(1);

if (cnt\_loop == 0)

fprintf(f, "Граф не содержит Гамильтоновых циклов\n");

else

{

for (i = 1; i <= cnt\_vertex; ++i)

print\_hamilton(f, i);

del\_loop();

}

}

fclose(f);

}

// Печать матрицы смежности

void print\_matrix(void)

{

int i, j, v;

char txt[20];

unsigned w;

// А есть ли граф?

if (empty\_graph())

return;

// Определим максимальную стоимость

for (i = v = 0; i < cnt\_vertex; ++i)

{

for (j = 0; j < cnt\_vertex; ++j)

{

if (mat[i][j] > v)

v = mat[i][j];

}

}

// Определим ширину максимального элемента

w = sprintf(txt, "%d", v);

printf("Матрица смежности\n");

for (i = 0; i < cnt\_vertex; ++i)

{

for (j = 0; j < cnt\_vertex; ++j)

{

printf("%\*d ", w, mat[i][j]);

}

printf("\n");

}

}

// Создание случайного графа

void rand\_graph(void)

{

int i, j, v1, v2, n, n1, n2;

if (!test\_num(param[1]))

{

printf("Количество вершин ошибочно\n");

return;

}

if ((n1 = atoi(param[1])) < 3 || n1 > MAX\_VERTEX)

{

printf("Количество вершин должно быть в диапазоне [3...%d]\n", MAX\_VERTEX);

return;

}

// Полный граф?

if (cnt == 2)

{

cnt\_vertex = n1;

for (i = 0; i < n1; ++i)

{

for (j = 0; j < n1; ++j)

{

if (i == j)

{

mat[i][j] = 0;

continue;

}

mat[i][j] = mat[j][i] = 5 + rand() % 20;

}

}

return;

}

if (!test\_num(param[2]))

{

printf("Количество ребер ошибочно\n");

return;

}

if ((n2 = atoi(param[2])) < 3 || n2 > (n1 \* (n1 - 1)) / 2)

{

printf("Количество ребер должно быть в диапазоне [3...%d]\n", (n1 \* (n1 - 1)) / 2);

return;

}

cnt\_vertex = n1;

for (i = 0; i < 100; ++i)

{

memset(mat, 0, sizeof(mat));

for (j = 0; j < n2; ++j)

{

for (;;)

{

v1 = rand() % n1;

v2 = rand() % n1;

if (v1 != v2 && mat[v1][v2] == 0)

{

mat[v1][v2] = mat[v2][v1] = 5 + rand() % 20;

break;

}

}

}

// Если граф односвязный - завершаем генерацию

connected\_parts();

if (cnt\_connect == 1)

break;

}

}

// Краткая справка

char help[] =

"\n"

"Help - краткая справка\n"

"Load name - загрузка графа из набора данных в файле с именем name\n"

"Save name - сохранение графа в файле с именем name\n"

"Manual vertex edge - создание графа вводом с консоли (vertex вершины , edge ребер)\n"

"Rand vertex edge - создание случайного графа (vertex вершин, edge ребер)\n"

"Matrix - печать матрицы смежности графа\n"

"Hamilton vertex - поиск Гамильтоновых циклов для вершины vertex\n"

"Exit - завершение работы\n"

"\n";

int main()

{

// Установить кодировку печатаемых символов

// для корректной печати и ввода русского текста

SetConsoleOutputCP(1251);

// Инициализация датчика случайных чисел

srand((unsigned)time(NULL));

// Печать справки

system("color 5F");

printf(help);

// Обработка команд

for (;;)

{

// Введенная команда ошибочна?

if (!select\_cmd())

continue;

// Выполняем команду

switch (command)

{

// Завершение выполнения программы

default:

return 0;

// Помощь

case HELP:

printf(help);

continue;

// Печать матрицы смежности

case MATRIX:

print\_matrix();

continue;

// Поиск Гамильтоновых циклов

case HAMILTON:

Hamilton();

continue;

// Получить описание графа из файла

case LOAD:

load\_graph();

continue;

// Получить описание графа с консоли

case MANUAL:

manual\_graph();

continue;

// Сохранить граф в файле

case SAVE:

save\_graph();

continue;

// Создать случайный граф

case RAND:

rand\_graph();

}

}

}