Частное учреждение образования

«Колледж бизнеса и права»

|  |  |
| --- | --- |
|  | УТВЕРЖДАЮ  Ведущий методист колледжа  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Е.В. Паскал  « \_\_\_\_ » \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2021 года |
| Специальность 2-40 01 01 «Программное обеспечение информационных технологий» | Учебная дисциплина «Основы алгоритмизации и программирование» |

**Лабораторная работа № 35**

**Инструкционно-технологическая карта**

Тема: Реализация алгоритмов поиска в глубину в графе, заданном различными способами.

Цель: Научиться создавать алгоритмы поиска в глубину в графе, заданном различными способами.

Время выполнения: 2 часа.

1. **Порядок выполнения работы**
2. Изучить теоретические сведения к лабораторной работе.
3. Разработать на языке С++ программу вывода на экран решения задачи в соответствии с вариантом индивидуального задания, указанным преподавателем.
4. Отлаженную, работающую программу сдать преподавателю. Работу программы показать с помощью самостоятельно разработанных тестов.
5. Ответить на контрольные вопросы.
6. **Теоретические сведения**

**Граф**

**Граф** – совокупность точек, соединенных линиями. Точки называются **вершинами**, или узлами, а линии – **ребрами**, или дугами.

**Степень входа вершины** – количество входящих в нее ребер, а **степень выхода вершины** – количество исходящих из этой вершины ребер.

Граф, содержащий ребра между всеми парами вершин, является **полным графом**.

Есть графы, ребрам которых поставлено в соответствие конкретное числовое значение (вес), они называются **взвешенными графами**, а это значение – **весом ребра**.

Когда у ребра оба конца совпадают, т.е. ребро выходит из вершины и входит в эту же вершину, то такое ребро называется **петлей**.

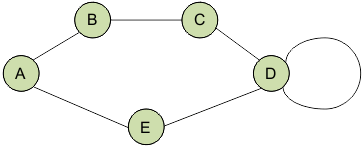


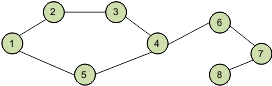
Рисунок 1 – Граф ABCDE с двунаправленными (неориентированными) ребрами и

петлей (ребро справа от вершины D)

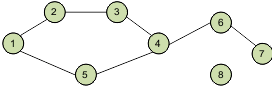
**Классификация графов**

Графы делятся на:

1. Связные



1. Несвязные

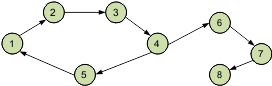


В **связном графе** между любой парой вершин существует как минимум один путь.

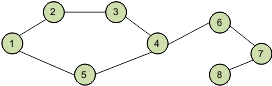
В **несвязном графе** существует хотя бы одна вершина, не связанная с другими вершинами этого графа (в примере вершина № 8).

Графы также подразделяются на:

1. Ориентированные



1. Неориентированные



1. Смешанные.

В **ориентированном графе** ребра являются направленными, т.е. существует только одно доступное направление движения между двумя связными вершинами. В **неориентированном графе** по каждому из ребер можно осуществлять переход в обоих направлениях. Частный случай двух этих видов – **смешанный граф**. Он содержит в себе как ориентированные, так и неориентированные ребра.

**Способы представления графа**

Граф может быть представлен (сохранен) несколькими способами:

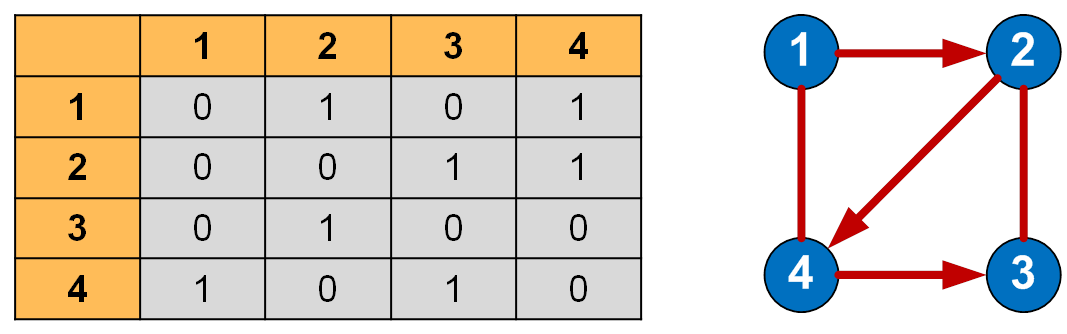
1. матрица смежности;
2. матрица инцидентности;
3. список смежности (инцидентности);
4. список ребер.

Использование двух первых методов предполагает хранение графа в виде двумерного массива (матрицы). Размер массива зависит от количества вершин и (или) ребер в конкретном графе.

**Матрица смежности графа** – это квадратная матрица, в которой каждый элемент принимает одно из двух значений: 0 или 1. Число строк матрицы смежности равно числу столбцов и соответствует количеству вершин графа:

0 – соответствует отсутствию ребра,

1 – соответствует наличию ребра.



Когда из одной вершины в другую проход свободен (имеется ребро), в ячейку заносится 1, иначе – 0. Все элементы на главной диагонали равны 0, если граф не имеет петель.

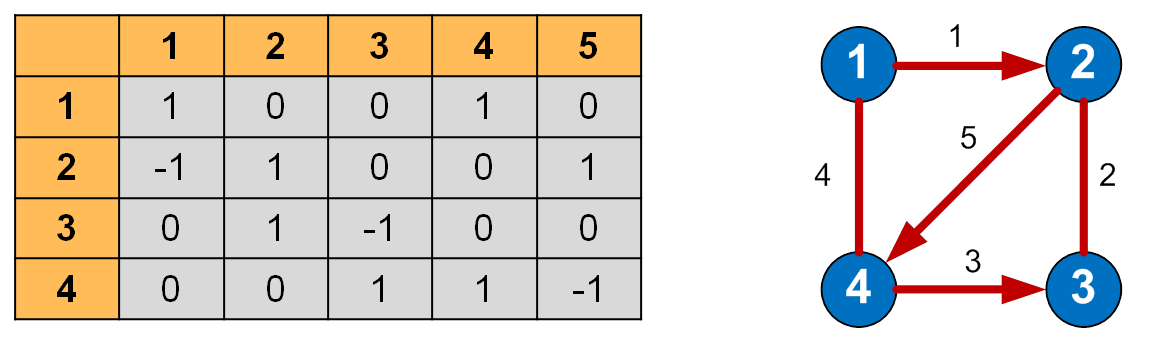
**Матрица инцидентности**

**Матрица инцидентности (инциденции) графа** – это матрица, количество строк в которой соответствует числу вершин, а количество столбцов – числу рёбер. В ней указываются связи между инцидентными элементами графа: ребро (дуга) и вершина.

В неориентированном графе, если вершина инцидентна ребру, то соответствующий элемент равен 1, в противном случае – элемент равен 0.

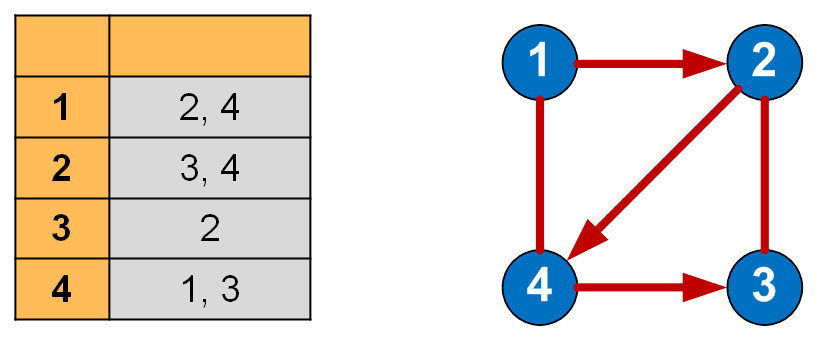
В ориентированном графе, если ребро выходит из вершины, то соответствующий элемент равен 1, если ребро входит в вершину, то соответствующий элемент равен -1, если ребро отсутствует, то элемент равен 0.

Матрица инцидентности для своего представления требует нумерации рёбер, что не всегда удобно.



**Список смежности (инцидентности)**

Если количество ребер графа по сравнению с количеством вершин невелико, то значения большинства элементов матрицы смежности будут равны 0. В таком случае использование данного метода нецелесообразно. Для подобных графов имеются более оптимальные способы их представления. По отношению к памяти компьютера **списки смежности** менее требовательны, чем матрицы смежности. Такой список можно представить в виде таблицы, столбцов в которой – 2, а строк – не больше, чем вершин в графе. В каждой строке в первом столбце указана вершина выхода, а во втором столбце – список вершин, в которые входят ребра из текущей вершины.



**Преимущества списка смежности**:

1. Рациональное использование памяти.
2. Позволяет быстро перебирать соседей у данной вершины.
3. Позволяет проверять наличие ребра и удалять его.

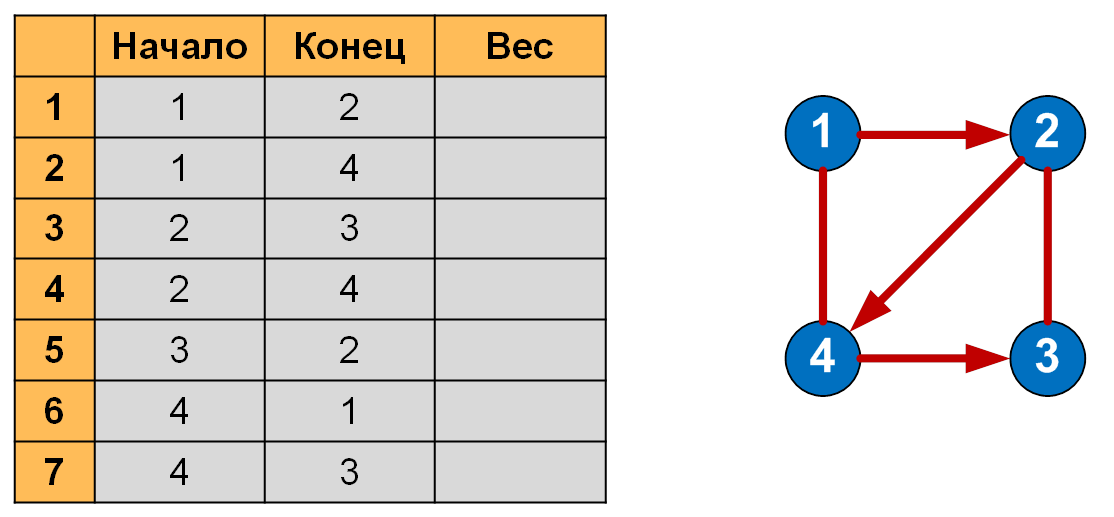
**Недостатки списка смежности**:

1. При работе с **насыщенными графами** (с большим количеством рёбер) скорости может не хватать для обработки всего большого графа.
2. Нет быстрого способа проверить, существует ли ребро между двумя вершинами.
3. Количество вершин графа должно быть известно заранее.
4. Для взвешенных графов приходится хранить список, элементы которого должны содержать два значащих поля (см. далее), что усложняет код:
   1. номер вершины, с которой соединяется текущая вершина;
   2. вес ребра (число).

**Список рёбер**

В **списке рёбер** в каждой строке записываются две смежные вершины и вес соединяющего их ребра (для взвешенного графа).

Количество строк в списке ребер всегда должно быть равно величине, получающейся в результате сложения ориентированных рёбер с удвоенным количеством неориентированных рёбер.



Какой способ представления графа лучше? Ответ зависит от отношения между числом вершин и числом рёбер. Число ребер может быть довольно малым (такого же порядка, как и количество вершин) или довольно большим (если граф является полным). Графы с большим числом рёбер называют **плотными**, с малым – **разреженными**. Плотные графы удобнее хранить в виде матрицы смежности, разреженные – в виде списка смежности.

**Алгоритмы обхода графов**

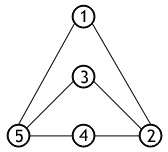
Основными алгоритмами обхода графов являются:

1. поиск в ширину;
2. поиск в глубину.

**Поиск в глубину** (англ. depth-first search, DFS) – это рекурсивный алгоритм обхода вершин графа. Если метод поиска в ширину производится симметрично (вершины графа просматриваются по уровням), то метод в глубину предполагает продвижение вглубь до тех пор, пока это возможно. Невозможность дальнейшего продвижения означает, что следующим шагом будет переход на последний узел, имеющий несколько вариантов движения (один из которых исследован полностью), ранее посещенный узел (вершину).

Отсутствие последнего свидетельствует об одной из двух возможных ситуаций: либо все вершины графа уже просмотрены, либо просмотрены все те, что доступны из вершины, взятой в качестве начальной, но не все (несвязные и ориентированные графы допускают последний вариант).

Рассмотрим то, как будет вести себя алгоритм на конкретном примере. Приведенный далее неориентированный связный граф имеет в сумме 5 вершин. Сначала необходимо выбрать начальную вершину. Какая бы вершина в качестве таковой не была выбрана, граф в любом случае исследуется полностью, поскольку, как уже было сказано, это связный граф без единого направленного ребра. Пусть обход начнется с узла 1, тогда порядок последовательности просмотренных узлов будет следующим: 1 2 3 5 4. Если выполнение начать, например, с узла 3, то порядок обхода окажется иным: 3 2 1 5 4.



Алгоритм поиска в глубину базируется на рекурсии, т.е. функция обхода, по мере выполнения, вызывает сама себя, что делает код в целом довольно компактным.

Обычно DFS (depth-first search) – название функции поиска в глубину в графе. Ее параметр st – стартовый узел, передаваемый из главной части программы как аргумент. Каждому из элементов логического массива visited заранее присваивается значение false, т.е. каждая из вершин изначально будет значиться как не посещенная (см. п. 3 пример кода).

Двумерный массив graph – это матрица смежности графа. Большую часть внимания следует сконцентрировать на последней строке. Если элемент матрицы смежности на каком-то шаге равен 1 (а не 0) и вершина с тем же номером, что и проверяемый столбец матрицы, при этом не была посещена, то функция рекурсивно повторяется. В противном случае функция возвращается на предыдущую стадию рекурсии. Для примера взят неориентированный граф, представленный матрицей смежности graph[5][5]. В коде этот двумерный массив реализован как динамический с прямой инициализацией при декларации (при создании).

**Алгоритм поиска (или обхода) в глубину** (англ. depth-first search, DFS) позволяет построить обход ориентированного или неориентированного графа, при котором посещаются все вершины, доступные из начальной вершины.

Отличие поиска в глубину от поиска в ширину заключается в том, что (в случае неориентированного графа) результатом алгоритма поиска в глубину является некоторый маршрут, следуя которому можно обойти последовательно все вершины графа, доступные из начальной вершины. Этим он принципиально отличается от поиска в ширину, где одновременно обрабатывается множество вершин; в поиске в глубину в каждый момент исполнения алгоритма обрабатывается только одна вершина. С другой стороны, поиск в глубину не находит кратчайших путей, зато он применим в ситуациях, когда граф неизвестен целиком, а исследуется каким-то автоматизированным устройством. Если же граф ориентированный, то поиск в глубину строит дерево путей из начальной вершины во все доступные из нее.

Обход в глубину можно представить себе следующим образом. Пусть исследователь находится в некотором лабиринте (графе) и он хочет обойти весь лабиринт (посетить все доступные вершины в графе). Исследователь находится в некоторой вершине и видит ребра, исходящие из этой вершины. Очевидная последовательность действий исследователя такая:

1. Пойти в какую-нибудь смежную вершину.
2. Обойти все, что доступно из этой вершины.
3. Вернуться в начальную вершину.
4. Повторить алгоритм для всех остальных вершин, смежных из начальной.

Видим, что алгоритм является рекурсивным – для обхода всего графа нужно переместиться в соседнюю вершину, после чего повторить для этой вершины алгоритм обхода. Но возникает проблема зацикливания – если из вершины A можно перейти в вершину B, то из вершины B можно перейти в вершину A и рекурсия будет бесконечной. Для борьбы с рекурсией нужно применить очень простую идею – исследователь не должен идти в ту вершину, в которой он уже был раньше, то есть которая не представляет для него интерес (считаем, что интерес для исследователя представляют только вершины, в которых он не был ранее). Итак, уточненный алгоритм может выглядеть следующим образом:

1. Пойти в какую-нибудь смежную вершину, не посещенную ранее.
2. Запустить из этой вершины алгоритм обхода в глубину.
3. Вернуться в начальную вершину.
4. Повторить пункты 1-3 для всех не посещенных ранее смежных вершин.

Для реализации алгоритма понадобится отмечать, в каких вершинах был исследователь, а в каких – нет. Пометку будем делать в списке visited, где visited[i] == true для посещенных вершин, и visited[i] == false для непосещенных. Пометка «о посещении вершины» ставится при заходе в эту вершину. Поскольку целью обхода в глубину зачастую является построение дерева обхода в глубину, то сразу же будем хранить предшественника для каждой вершины. Алгоритм обхода в глубину оформим в виде рекурсивной функции dfs, где start – номер вершины, из которой запускается обход. Пример реализации на языке C++:

void dfs(int start, vector<bool> & visited, vector <int> & prev, const vector <vector <int> > g)

{

visited[start] = true;

for (auto u : g[start])

{

if (!visited[u])

{

prev[u] = start;

dfs(u, visited, prev, g);

}

}

}

int main()

{//сначала надо создать для графа динамические массивы vector <типДанных> имя

vector <bool> visited(n + 1);

vector <int> prev(n + 1, -1);

dfs(start, visited, prev, g);

}

В этом алгоритме n – число вершин в графе, вершины нумеруются числами от 1 до n, а v[u] хранит множество вершин смежных с u. Для запуска алгоритма, например, для вершины с номером start необходимо вызвать dfs. После этого вызова все вершины, доступные из start, будут отмечены в списке visited, а при помощи списка prev можно построить пути из вершины start до всех доступных вершин. Если не требуется строить дерево обхода в глубину, то можно убрать заполнение списка start, в этом случае алгоритм dfs проще.

**Выделение компонент связности**

Алгоритм обхода в глубину позволяет решать множество различных задач. Например, реализуем при помощи алгоритма обхода в глубину подсчет числа **компонент связности** в неориентированном графе. Для этого будем обходить все вершины графа и проверять, была ли очередная вершина посещена ранее. Если не была – то это означает, что найдена новая компонента связности, для выделения всей компоненты связности необходимо запустить DFS от этой вершины. Пример реализации на языке C++:

void dfs(int start, vector<bool> & visited, const vector <vector <int> > g)

{

visited[start] = true;

for (auto u : g[start])

{

if (!visited[u])

{

dfs(u, visited, g);

}

}

}

int main()

{//сначала надо создать для графа динамические массивы vector <типДанных> имя

vector <bool> visited(n + 1);

int ncomp = 0;

for (int i = 1; i <= n; ++i)

{

if (!visited[i])

{

++ncomp;

dfs(start, visited, g);

}

}

}

**Проверка графа на двудольность**

Граф называется **двудольным**, если его вершины можно разбить на два множества так, что концы каждого ребра принадлежат разным множествам. Иными словами, можно покрасить вершины графа в два цвета так, что концы каждого ребра покрашены в разный цвет. Модифицируем алгоритм DFS так, что он будет проверять граф на двудольность и строить покраску графа в два цвета (если он двудольный). Для этого заменим список Visited на список Color, в котором будем хранить значение 0 для непосещенных вершин, а для посещенных вершин будем гранить значение 1 или 2 – ее цвет.

Алгоритм DFS для каждого ребра будет проверять цвет конечной вершины этого ребра. Если вершина не была посещена, то она красится в цвет, неравный цвету текущей вершины. Если же вершина была посещена, то ребро либо пропускается, если его концы – разноцветные, а если его концы одного цвета, то делается пометка, что граф не является двудольным (переменной IsBipartite присваивается значение False, по ее значению можно судить о том, является ли граф двудольный). Пример реализации на языке C++:

bool is\_bipartite = true;

void dfs(int start, vector<int> & color, const vector <vector <int> > g)

{

for (auto u : g[start])

{

if (color[u] == 0)

{

color[u] = 3 - color[start]

dfs(u, color, g);

}

else

{

if (color[u] == color[start])

{

is\_bipartite = false;

}

}

}

int main()

{//сначала надо создать для графа динамические массивы vector <типДанных> имя

vector <int> color(n + 1);

for (int i = 1; i <= n; ++i)

{

if (color[i] == 0)

{

color[i] = 1;

dfs(i, color, g);

}

}

}

Основная программа проходит по всем ребрам графа и при обнаружении ранее не обнаруженной вершины красит ее в цвет 1 и запускает DFS из этой вершины.

**Поиск цикла в ориентированном графе**

Цикл в ориентированном графе можно обнаружить по наличию ребра, ведущего из текущей вершины в вершину, которая в настоящий момент находится в стадии обработки, то есть алгоритм DFS зашел в такую вершину, но еще не вышел из нее. В таком алгоритме DFS будем красить вершины в три цвета. Цветом 0 («белый») будем обозначать еще непосещенные вершины. Цветом 1 («серый») будем обозначать вершины в процессе обработки, а цветом 2 («черный») будем обозначать уже обработанные вершины. Вершина красится в цвет 1 при заходе в эту вершину и в цвет 2 – при выходе. Цикл в графе существует, если алгоритм DFS обнаруживает ребро, конец которого покрашен в цвет 1. Пример реализации на языке C++:

bool cycle\_found = false;

void dfs(int start, vector<int> & color, const vector <vector <int> > g)

{

color[start] = 1;

for (auto u : g[start])

{

if (color[u] == 0)

{

dfs(u, color, g);

}

else

{

if (color[start] == 1)

{

cycle\_found = true;

}

}

}

color[start] = 2;

}

int main()

{//сначала надо создать для графа динамические массивы vector <типДанных> имя

vector <int> color(n + 1);

for (int i = 1; i <= n; ++i)

{

if (color[i] == 0)

{

dfs(start, color, g);

}

}

}

**Топологическая сортировка**

Наконец, еще одно важное применение поиска в глубину – **топологическая сортировка**. Пусть дан ориентированный граф, не содержащий циклов. Тогда вершины этого графа можно упорядочить так, что все ребра будут идти от вершин с меньшим номером к вершинам с большим номером. Для топологической сортировки графа достаточно запустить алгоритм DFS, при выходе из вершины добавляя вершину в конец списка с ответом. После окончания алгоритма список с ответом развернуть в противоположном порядке. Пример реализации на языке C++:

void dfs(int start, vector <bool> & visited, vector <bool> & ans, const vector <vector <int> > g)

{

visited[start] = true;

for (auto u : g[start])

{

if (!visited[u])

{

dfs(u, visited, ans, g);

}

}

ans.push\_back(start);

}

int main()

{//сначала надо создать для графа динамические массивы vector <типДанных> имя

vector <bool> visited(n + 1);

vector <int> ans;

for (int i = 1; i <= n; ++i)

{

if (!visited[i])

{

dfs(start, visited, ans, g);

}

}

reverse(ans.begin(), ans.end());

}

**Поиск мостов**

**Мостом** называется ребро, при удалении которого граф распадается на две компоненты связности. Алгоритм поиска в глубину позволяет найти все мосты в связном графе за один DFS, то есть за сложность O(n).

Подвесим граф за какую-то вершину, запустим из этой вершины DFS. DFS построит дерево обхода графа, при этом будут найдены **обратные рёбра** – рёбра, которые идут из текущей вершины в вершину, которая находится в настоящий момент в стадии обработки. Каждой вершине u сопоставим значение h(u) – её глубина в дереве обхода. Кроме этого, каждой вершине сопоставим значение функции f(u), где f(u) – это минимальное значение h(v) для всех вершин v, которые достижимы из вершины u в дереве обхода, а также достижимы при помощи прохода по одному обратному ребру из любого потомка u в дереве обхода. Тогда ребро uv будет мостом, если f(v) > h(u). Значения h(u) и f(u) можно считать одним DFS. Пример реализации на языке C++:

void dfs(int u, int parent, int curr\_h, vector <vector<int> > & g, vector <bool> & visited, vector<int> & h, vector<int> & f)

{

h[u] = curr\_h++;

f[u] = h[u];

visited[u] = true;

for (auto v : g[u])

{

if (v == parent)

{

continue;

}

if (!visited[v])

{

dfs(v, u, curr\_h, g, visited, h, f);

f[u] = min(f[u], f[v]);

if (f[v] > h[u])

{

// Найден мост u-v

}

}

else

{

f[u] = min(f[u], h[v]);

}

}

}

Параметры, передаваемые в функцию:

u – текущая вершина

parent – родитель, чтобы не проходить по ребру в обратном направлении (эта реализация не работает на графе с кратными ребрами).

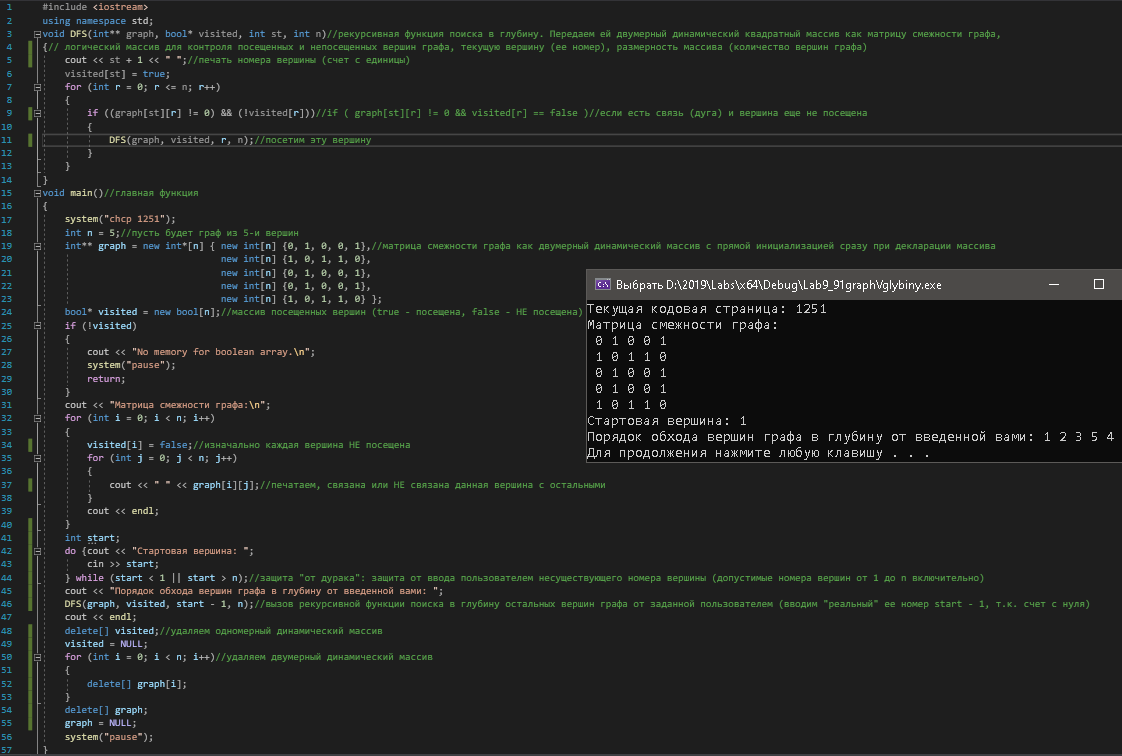
curr\_h – текущая глубина

g – списки смежности графа

h – массив значений глубины для вершин

f – массив значения целевой функции для вершин

1. **Пример выполнения программы**



1. **Задания по вариантам**

Вывести матрицы смежности, инцидентности, весов для графа, представленного на рисунке по вашему варианту. Реализовать алгоритм поиска в глубину значения вершины графа от значения элемента, вводимого пользователем с клавиатуры, в вашем графе.

1)

**1**

**2**

**8**

**3**

**42**

**5**

**9**

**7**

**6**

2)

**2**

**3**

**1**

**6**

**5**

**7**

**8**

**4**

3)

**1**

**2**

**3**

**42**

**5**

**6**

**7**

**8**

**9**

4)

**1**

**2**

**8**

**1**

**72**

**39**

**91**

**49**

**42**

**83**

5)

**2**

**3**

**1**

**6**

**8**

**5**

**4**

**7**

6)

**3**

**2**

**1**

**6**

**5**

**7**

**8**

**4**

7)

**7**

**3**

**1**

2

**42**

**5**

**6**

**8**

**9**

8)

**1**

**3**

**5**

**42**

**2**

**6**

**7**

**8**

**9**

9)

**1**

**2**

6

**42**

**5**

**9**

**7**

**3**

8

10)

**2**

**3**

**1**

**6**

**5**

**7**

**8**

**4**

11)

**4**

3

**62**

**5**

**2**

**7**

**8**

**9**

1

12)

**5**

**2**

**8**

**42**

**1**

**6**

**7**

**3**

**9**

13)

**2**

**3**

**1**

**6**

**5**

**7**

**8**

**4**

14)

**2**

**3**

**1**

**6**

**5**

**7**

**8**

**4**

15)

**2**

**3**

**1**

**6**

**5**

**7**

**8**

**4**

1. **Контрольные вопросы**
2. Дайте определение понятию «граф».
3. Дайте определение понятию «ребро графа».
4. Дайте определение понятию «полный граф»?
5. Дайте определение понятию «смешанный граф»?
6. Дайте определение понятию «взвешенный граф»?
7. Что такое «вес ребра»?
8. В чем разница между связным и несвязным графами?
9. В чем разница между ориентированным и неориентированным графами?
10. Что такое матрица смежности и как в нее сохранить данные графа?
11. Что такое матрица инцидентности и как в нее сохранить данные графа?
12. Что такое список смежности (инцидентности) и как в него сохранить данные графа?
13. Что такое список ребер и как в него сохранить данные графа?
14. В чем преимущества и недостатки списка смежности для графа?
15. В чем разница полного и насыщенного графов?
16. В чем разница плотного и разреженного графов?
17. Какие есть способы обхода графов?
18. Опишите поиск в глубину для графа.
19. Для решения каких задач подходит обход в глубину для графа?
20. Что такое компонента связности?
21. Что такое двудольность графа?
22. Что такое «цикл в ориентированном графе»?
23. Что такое топологическая сортировка в графе?
24. Что такое «мост в графе» и как его найти?
25. Что такое обратное ребро?

**Литература**

**Дейтел,** Х.М. Как программировать на С++ / Х.М. Дейтел, П.Дж. Дейтел . – М. : Бином-Пресс , 2018 . – 1456 с.

**Павловская**, Т.А. С++. Объектно-ориентированное программирование : практикум / Т.А. Павловская, Ю.А. Щупак . – СПб. : Питер , 2019 . – 265 с.

**Страуструп**, Б. Язык программирования С++ / Б. Страуструп . – СПб. : Бином-Пресс , 2019 . – 1054 с.

Преподаватель Шаляпин Ю.В.

|  |
| --- |
| Рассмотрено на заседании цикловой  комиссии ПОИТ № 10  Протокол №\_\_\_\_от «\_\_\_»\_\_\_\_\_\_\_\_\_\_2021 г.  Председатель ЦК \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |