## Основные термины теории графов

Многие объекты, возникающие в жизни человека, могут быть смоделированы (представлены в памяти компьютера) при помощи графов. Например, транспортные схемы (схема метрополитена и т. д.) изображают в виде станций, соединенных линиями. В терминах графов станции называются вершинами графа а линии – ребра.

**Графом** называется конечное множество вершин и множество ребер. Каждому ребру сопоставлены две вершины – концы ребра.

Бывают различные варианты определения графа. В данном определении концы у каждого ребра – равноправны. В этом случае нет разницы где начало, а где – конец у ребра. Но, например, в транспортных сетях бывают случаи одностороннего движения по ребру, тогда говорят об **ориентированном** графе – графе, у ребер которого одна вершина считается начальной, а другая – конечной.  
Если некоторое ребро u соединяет две вершины A и B графа, то говорят, что ребро u **инцидентно** вершинам A и B, а вершины в свою очередь инцидентны ребру u. Вершины, соединенные ребром, называются **смежными**.

Ребра называются **кратными**, если они соединяют одну и ту же пару вершин (а в случае ориентированного графа – если у них совпадают начала и концы). Ребро называется **петлей**, если у него совпадают начало и конец. Во многих задачах кратные ребра и петли не представляют интереса, поэтому могут рассматриваться только графы без петель и кратных ребер. Такие графы называю **простыми**.

Степенью вершины в неориентированном графе называется число инцидентных данной вершине ребер (при этом петля считается два раза, то есть степень - это количество «концов» ребер, входящих в вершину). Довольно очевидно, что сумма степеней всех вершин равна удвоенному числу ребер в графе. Отсюда можно посчитать максимальное число ребер в простом графе - если у графа n вершин, то степень каждой из них равна n−1, а, значит, число ребер есть n(n−1)/2. Граф, в котором любые две вершины соединены одним ребром, называется полным **графом**.

Также легко заметить следующий факт – в любом графе число вершин нечетной степени – четно. Этот факт называется **«леммой о рукопожатиях»** – в любой компании число людей, сделавших нечетное число рукопожатий всегда четно.

## Пути, циклы, компоненты связности

**Путем** на графе называется последовательность ребер u1, u2, …, uk, в которой конец одного ребра является началом следующего ребра. Начало первого ребра называется началом пути, конец последнего ребра - концом пути. Если начало и конец пути совпадают, то такой путь называется **циклом**.

Путь, который проходит через каждую вершину не более одного раза называется простым путем. Аналогично определяется **простой цикл**.

Граф называется **связным**, если между любыми двумя его вершинами есть путь. Если граф несвязный, то его можно разбить на несколько частей (подграфов), каждая из которых будет связной. Такие части называются **компонентами связности**. Возможно, что некоторые компоненты связности будут состоять всего лишь из одной вершины.

Понятно, что в графе из n вершин может быть от 1 до n компонент связности.

## Деревья

Рассмотрим связный граф из n вершин. Какое минимальное число ребер может быть в нем?

Несложно построить пример графа, содержащего n−1 ребро – например, можно взять одну вершину графа и соединить ее с n−1 ребром. Нетрудно также понять, что в таком графе не должно быть простых циклов (иначе в простом цикле можно выбросить одно ребро и граф останется связным). Такие графы называются **деревьями**.

Определение – **деревом** называется связный граф не содержащий простых циклов.

Нетрудно видеть, что в дереве нельзя удалить ни одного ребра, чтобы граф остался связным. Поэтому дерево является минимальным связным графом.

Основным свойством дерева является следующая теорема:

Дерево из n вершин содержит n−1 ребро.

Эту теорему можно доказать математической индукцией по n, используя лемму о висячей вершине – в каждом дереве есть хотя бы одна вершина степени 1. Эту вершину можно удалить и далее применить предположение индукции для меньшего числа вершин.

Можно показать, что эквивалентны следующие определения дерева:

1. Деревом называется связный граф не содержащий простых циклов.
2. Деревом называется связный граф, содержащий n вершин и n−1 ребро.
3. Деревом называется связный граф, который при удалении любого ребра перестает быть связным.
4. Деревом называется граф, в котором любые две вершины соединены ровно одним простым путем.

Очень часто в дереве выделяется одна вершина, называемая корнем дерева, дерево с выделенным корнем называют корневым или подвешенным деревом. Примером такого дерева является генеалогическое дерево.

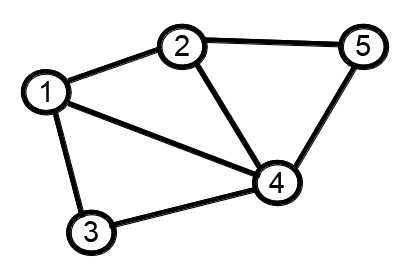
## Способы представления графов в памяти

Представление графов в памяти – это способ хранения информации о ребрах графа, позволяющий решать следующие задачи:

1. Для двух данных вершин u и b проверить, соединены ли вершины u и v ребром.
2. Перебрать все ребра, исходящие из данной вершины u .

При этом способ хранения графов в памяти должен учитывать возможности работы с ориентированными и неориентированными графами. По умолчанию будем предполагать, что хранимый граф является простым, но можно рассмотреть вопрос и о представлении графов с петлями и кратными ребрами.

Рассмотрим следующий граф:



При представлении графа **матрицей смежности** информация о ребрах графа хранится в квадратной матрице (двумерном списке), где элемент A[i][j] равен1, если ребра i и j соединены ребром и равен 0 в противном случае. Для данного примера матрица смежности будет выглядеть так:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 |
| 1 | 0 | 1 | 1 | 1 | 0 |
| 2 | 1 | 0 | 0 | 1 | 1 |
| 3 | 1 | 0 | 0 | 1 | 0 |
| 4 | 1 | 1 | 1 | 0 | 1 |
| 5 | 0 | 1 | 0 | 1 | 0 |

Если граф неориентированный, то матрица смежности всегда симметрична относительно главной диагонали.  
При использовании матрицы смежности удобно проверять соединены ли две вершины ребром – это просмотр одного элемента матрицы A[i][j], но сложнее перебирать все ребра, исходящие из данной вершины (для этого необходимо перебрать все оставшиеся вершины и проверить, соединены ли они ребром). Также матрица смежности требует O(n2) памяти и может оказаться неэффективным способом хранения дерева или разреженных графов.

При представлении графа **списками смежности** для каждой вершины i хранится список W[i] смежных с ней вершин. Для рассмотренного примера списки будут такими:

W[1] = [2, 3, 4]  
W[2] = [1, 4, 5]  
W[3] = [1, 4]  
W[4] = [1, 2, 3, 5]  
W[5] = [2, 4]

Таким образом, весь граф можно представить одним списком, состоящим из вложенных списков смежности вершин.

W = [[], [2, 3, 4], [1, 4, 5], [1, 4], [1, 2, 3, 5], [2, 4]]

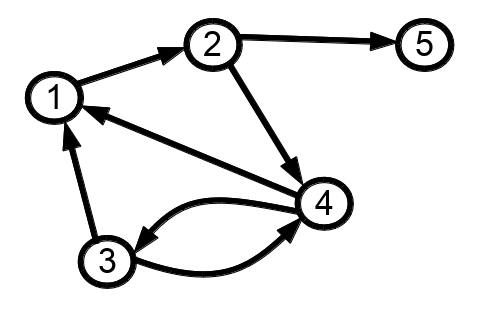
Поскольку нумерация в нашем примере начинается с 0, то к списку добавлен еще один фиктивный элемент W[0].

В языке С++ для представления списков смежности будем использовать тип vector<vector<int> >, то есть массив (вектор), элементами которого являются динамические массивы (векторы) чисел. В языке Python будут использоваться списки, элементами которого являются списки смежности.

В таком способе удобно перебирать ребра, выходящие из вершины i (это просто список W[i]), но сложно проверять наличие ребра между вершинами i и j – для этого необходимо проверить, содержится ли число j в списке W[i]. Но в языке Python можно эту часть сделать более эффективной, если заменить списки на множества – тогда проверка существования ребра между двумя вершинами также будет выполняться за O(1).

При помощи матриц смежности и списков смежности можно представлять и неориентированные графы. В случае матрицы смежности A[i][j] будет равно 1, если есть ребро, начинающееся в вершине i и заканчивающееся в вершине j. В случае списков смежности наличие ребра из вершины i в вершину j означает, что в списке W[i] есть число j.

Например, для такого графа:



Матрица смежностей будет следующей:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 |
| 1 | 0 | 1 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 1 | 1 |
| 3 | 1 | 0 | 0 | 1 | 0 |
| 4 | 1 | 0 | 1 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 |

А списки смежности будут следующими:

W[1] = [2]  
W[2] = [4, 5]  
W[3] = [1, 4]

W[4] = [1, 3]  
W[5] = []

Приведем код считывания графа. Будем считать, что граф задается следующим образом: в первой строке записано число вершин n и число ребер m графа. Далее записаны m строк, содержащих по два числа – номера начальной и конечной вершины ребра. Например, первый граф из первого примера можно задать так:

5 7  
1 2  
2 5  
5 4  
4 2  
1 4  
1 3  
3 4

Пример заполнения матрицы смежности. Матрица создается размером (n+1)×(n+1) , так как используется нумерация с единицы:

**C++**

cin >> n >> m;  
vector <vector<int> > A(n + 1, vector<int>(n + 1));  
for (int i = 0; i < m; ++ i) {  
    int u, v;  
    cin >> u >> v;  
    A[u][v] = 1;  
    // A[v][u] = 1;  
}

Пример заполнения списков смежности, используются множества вместо списков:

**C++**

cin >> n >> m;  
vector <vector<int> > W(n + 1);  
for (int i = 0; i < m; ++ i) {  
    int u, v;  
    cin >> u >> v;  
    W[u].push\_back(v);  
    // W[v].push\_back(u);  
}

Здесь также используется нумерация с единицы. Во всех примерах закомментированная строчка нужна в случае неориентированного графа, тогда для каждого считанного ребра из u в v необходимо добавить обратное ребро из v в  u.

## Взвешенные графы

Очень часто рассматриваются графы, в которых каждому ребру приписана некоторая числовая характеристика – **вес**. Вес может означать длину дороги или стоимость проезда по данному маршруту. Соответствующие графы называются взвешенными.

При представлении графа матрицей смежности вес ребра можно хранить в матрице, то есть A[i][j] в данном случае будет равно весу ребра из i в j. При этом при отсутствии ребра можно хранить специальное значение, например, None. Во многих задачах удобно при отсутствии ребра хранить очень большое число, в этом случае отсутствие ребра аналогично наличию ребра очень большой стоимости.

При представлении графа списками смежности можно поступить двумя способами. Можно в списках смежности хранить пару (кортеж) из двух элементов – номер конечной вершины и вес ребра. Но в этом случае неудобно проверять наличие ребра между двумя вершинами.

Другой способ – хранить списки смежности как ранее, а веса ребер хранить в отдельном ассоциативном массиве (map в C++, dict в Python), в котором ключом будет пара из двух номеров вершин (номер начальной и конечной вершины), а значением будет вес ребра между этими вершинами.

## СПОСОБЫ ПРЕДСТАВЛЕНИЯ ГРАФОВ В ПАМЯТИ

Представление графов в памяти — это способ хранения информации о ребрах графа, позволяющий решать следующие задачи:

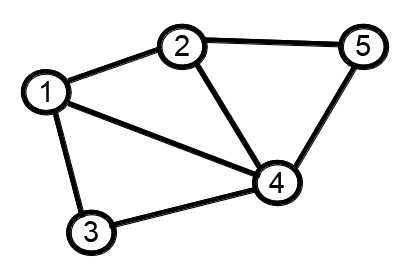
1. Для двух данных вершин u и v проверить, соединены ли вершины u и v ребром.
2. Перебрать все ребра, исходящие из данной вершины u .

При этом способ хранения графов в памяти должен учитывать возможности работы с ориентированными и неориентированными графами. По умолчанию будем предполагать, что хранимый граф является простым, но можно рассмотреть вопрос и о представлении графов с петлями и кратными ребрами.

Основными способами хранения графа являются

1. Матрица смежности.
2. Списки (или множества) смежных вершин.
3. Список ребер.

Рассмотрим следующий граф:



При представлении графа **матрицей смежности** информация о ребрах графа хранится в квадратной матрице (двумерном списке), где элемент A[i][j] равен 1, если ребра i и j соединены ребром и равен 0 в противном случае. Для данного примера матрица смежности будет выглядеть так:

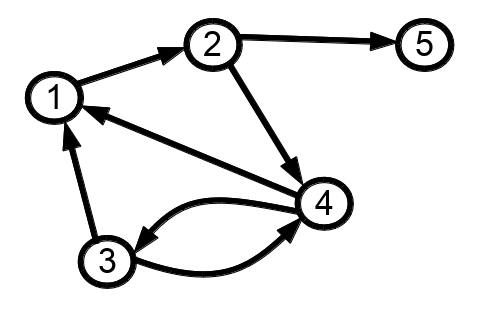
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 |
| 1 | 0 | 1 | 1 | 1 | 0 |
| 2 | 1 | 0 | 0 | 1 | 1 |
| 3 | 1 | 0 | 0 | 1 | 0 |
| 4 | 1 | 1 | 1 | 0 | 1 |
| 5 | 0 | 1 | 0 | 1 | 0 |

Если граф неориентированный, то матрица смежности всегда симметрична относительно главной диагонали.  
При использовании матрицы смежности удобно проверять соединены ли две вершины ребром – это просмотр одного элемента матрицы

A[i][j], но сложнее перебирать все ребра, исходящие из данной вершины (для этого необходимо перебрать все оставшиеся вершины и проверить, соединены ли они ребром). Также матрица смежности требует O(n2) памяти и может оказаться неэффективным способом хранения дерева или разреженных графов.

При помощи матриц смежности можно представлять и неориентированные графы. В случае матрицы смежности A[i][j] будет равно 1, если есть ребро, начинающееся в вершине i и заканчивающееся в вершине j.

Например, для такого графа:



Матрица смежнoстей будет следующей:

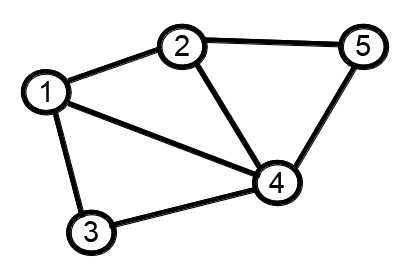
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 |
| 1 | 0 | 1 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 1 | 1 |
| 3 | 1 | 0 | 0 | 1 | 0 |
| 4 | 1 | 0 | 1 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 |

## ВЗВЕШЕННЫЕ ГРАФЫ

Очень часто рассматриваются графы, в которых каждому ребру приписана некоторая числовая характеристика – **вес**. Вес может означать длину дороги или стоимость проезда по данному маршруту. Соответствующие графы называются взвешенными.

При представлении графа матрицей смежности вес ребра можно хранить в матрице, то есть A[i][j] в данном случае будет равно весу ребра из i в j. При этом при отсутствии ребра можно хранить специальное значение, например, None в Python. Во многих задачах удобно при отсутствии ребра хранить очень большое число, в этом случае отсутствие ребра аналогично наличию ребра очень большой стоимости.

Рассмотрим следующий граф:



При представлении графа **списком ребер** для каждого ребра графа хранится пара номеров вершин, которые оно соединяет. Так, граф на рисунке задается таким списком пар:

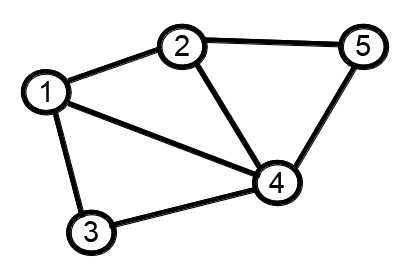
[1,2]  
[1,3]  
[1,4]  
[2,5]  
[4,5]  
[2,4]  
[3,4]

Для графа с Е ребрами такой способ хранения требует порядка Е памяти, но при этом основные операции — проверка наличия ребра между двумя вершинами и перебор вершин, смежных с данной — выполняются также за время порядка Е, поскольку требуют полного перебора всех списков ребер.

Иногда удобно хранить каждое ребро неориентированного графа дважды: как [u, v] и как [v, u].

**Для ориентированных графов** первое число в паре соответствует начальной вершине ребра, а второе — конечной.

Рассмотрим следующий граф:



При представлении графа **списками смежности** для каждой вершины i хранится список W[i] смежных с ней вершин. Для рассмотренного примера:

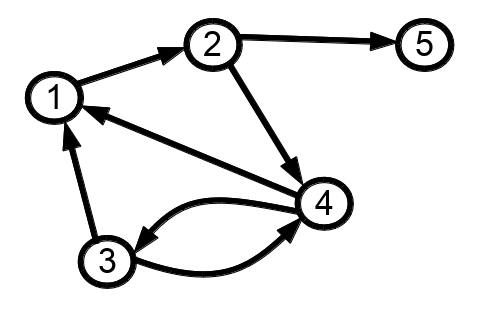
W[1] = [2, 3, 4]   
W[2] = [1, 4, 5]   
W[3] = [1, 4]   
W[4] = [1, 2, 3, 5]

W[5] = [2, 4]

В таком способе удобно перебирать ребра, выходящие из вершины i (это просто список W[i]), но сложно проверять наличие ребра между вершинами i и j – для этого необходимо проверить, содержится ли число j в списке W[i]. Но если заменить списки на множества, то и эта проверка существования ребра между двумя вершинами также будет выполняться за O(1).

При помощи списков смежности можно представлять и неориентированные графы.

Например, для такого графа:



списки смежности будут следующими:

W[1] = [2]   
W[2] = [4, 5]   
W[3] = [1, 4]   
W[4] = [1, 3]   
W[5] = []

ВЗВЕШЕННЫЕ ГРАФЫ

Очень часто рассматриваются графы, в которых каждому ребру приписана некоторая числовая характеристика – **вес**. Вес может означать длину дороги или стоимость проезда по данному маршруту. Соответствующие графы называются взвешенными.

При представлении графа списками смежности можно поступить двумя способами. Можно в списках смежности хранить кортеж из двух элементов – номер конечной вершины и вес ребра. Но в этом случае неудобно проверять наличие ребра между двумя вершинами.

Другой способ – хранить списки смежности как ранее, а веса ребер хранить в отдельном ассоциативном массиве, в котором ключом будет пара из двух номеров вершин (номер начальной и конечной вершины), а значением будет вес ребра между этими вершинами.

**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 вершиных» 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()  
{  
    …  
    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 становится чрезвычайно простым.

# 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()  
{  
    …  
    vector <bool> visited(n + 1);  
    int ncomp = 0;  
    for (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);  
}

int main()  
{  
    …  
    vector <int> color(n + 1);  
    for (i = 1; i <= n; ++i)  
        if (color[i] == 0) {  
            ++ncomp;  
            dfs(start, 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()  
{  
    …  
    vector <int> color(n + 1);  
    for (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()  
{  
    …  
    vector <bool> visited(n + 1);  
    vector <int> ans;  
    for (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 uсопоставим 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. #include <iostream>
2. #include <vector>
4. using namespace std;
6. int i, j;

9. void DFS(int st, int n, vector<vector<int>> &graph, bool visited[])
10. {
11. int r;
12. visited[st]=true;
13. for (r=0; r<=n; r++)
14. if ((graph[st][r]!=0) && (!visited[r]))
15. DFS(r, n, graph, visited);
16. }
18. int main()
19. {
20. int n;
21. cin >> n;
22. bool \*visited=new bool[n];
24. vector<vector<int>> graph(n, vector<int>(n, 0));


28. for(i=0; i<n; i++){="" int="" x;="" cin="">> x;
29. if(x>0){
30. graph[x-1][i] = 1;
31. }
32. }

35. for (i=0; i<n; i++)="" {="" visited[i]="false;" }="" bool="" \*vis="new" bool[n];="" int="" outlist[n];="" for(i="0;i<n;i++){" s="0;" dfs(i,n,="" graph,="" visited);="" for(j="0;j<n;j++){" if(visited[j]="=" true){="" s++;="" visited[j]="false;" outlist[i]="s;" max="0;" maxi;="" i<n;="" i++){="" if(outlist[i]=""> max) {
36. max = outlist[i];
37. maxi = i+1;
38. }
39. }
41. cout << maxi << " " << max;
43. return 0;
44. }
45. </n;></n;></int></vector<int></vector<int></vector></iostream>

***DFS***

#include <iostream>

#include <vector>

using namespace std;

int i, j;

void DFS(int st, int n, vector<vector<int>> &graph, bool visited[])

{

int r;

visited[st]=true;

for (r=0; r<=n; r++)

if ((graph[st][r]!=0) && (!visited[r]))

DFS(r, n, graph, visited);

}

int main()

{

int n;

cin >> n;

bool \*visited=new bool[n];

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

for(i=0; i<n; i++){="" int="" x;="" cin="">> x;

if(x>0){

graph[x-1][i] = 1;

}

}

for (i=0; i<n; i++)="" {="" visited[i]="false;" }="" bool="" \*vis="new" bool[n];="" int="" outlist[n];="" for(i="0;i<n;i++){" s="0;" dfs(i,n,="" graph,="" visited);="" for(j="0;j<n;j++){" if(visited[j]="=" true){="" s++;="" visited[j]="false;" outlist[i]="s;" max="0;" maxi;="" i<n;="" i++){="" if(outlist[i]=""> max) {

max = outlist[i];

maxi = i+1;

}

}

cout << maxi << " " << max;

return 0;

}

</n;></n;></int></vector<int></vector<int></vector></iostream>

в функции DFS этот for (r=0; r<=n; r++)

цикл делает одну лишнюю итерацию выходя за рамки как graph так и массива visited - непонятно почему вы там поставили знак равно !?

поменяйте на

for (r=0; r < n; r++)

так как индексация с нуля, значит последний элемент будет под номером n - 1