

Теория графов

Информатика. Язык программирования Python в курсе информатики. Все классы.

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.

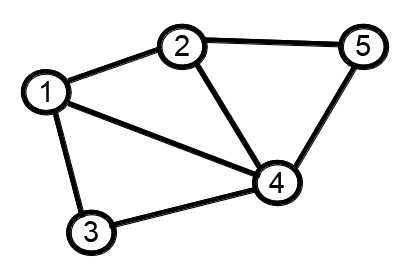
Способы представления графов в памяти

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]]

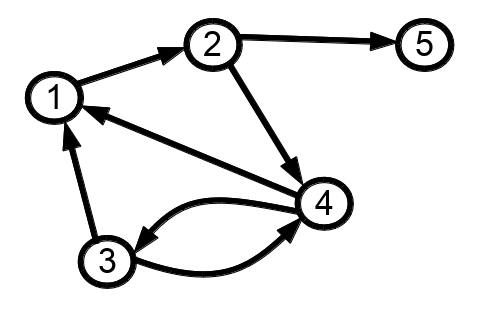
Поскольку нумерация в нашем примере начинается с 0, то к списку добавлен еще один фиктивный элемент W[0].

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;  
}

**Python**

n, m = map(int, input().split())  
A = [[0] \* (n + 1) for i in range(n + 1)]  
for i in range(m):  
    u, v = map(int, input().split())  
    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);  
}

**Python**

n, m = map(int, input().split())  
W = [set() for i in range(n + 1)]  
for i in range(m):  
    u, v = map(int, input().split())  
    W[u].add(v)  
    # W[v].add(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 хранения графа

Информатика. Язык программирования Python в курсе информатики. Все классы.

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.



Алгоритм поиска в глубину

Информатика. Язык программирования Python в курсе информатики. Все классы.

**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);

ПРИМЕР РЕАЛИЗАЦИИ НА ЯЗЫКЕ PYTHON

visited = [False] \* (n + 1)  
prev = [None] \* (n + 1)  
  
def dfs(start, visited, prev, g):  
    visited[start] = True  
    for u in g[start]:  
        if not visited[u]:  
            prev[u] = start   
            dfs(u)  
  
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);  
        }

ПРИМЕР РЕАЛИЗАЦИИ НА ЯЗЫКЕ PYTHON

Visited = [False] \* (n + 1)  
  
def DFS(start):  
    Visited[start] = True  
    for v in V[start]:  
        if not Visited[v]:  
            DFS(v)  
  
ncomp = 0  
for i in range(1, n + 1):   
    if not Visited(i):  
        ncomp += 1  
        DFS(i)

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);  
        }

ПРИМЕР РЕАЛИЗАЦИИ НА ЯЗЫКЕ PYTHON

Color = [0] \* (n + 1)  
IsBipartite = True  
  
def DFS(start):  
    for u in V[start]:  
        if Color[u] == 0:  
            Color[u] = 3 - Color[start]  
            DFS(u)   
        else if Color[u] == Color[start]:  
            IsBipartite = False   
  
for i in range(1, n + 1):   
    if Color[i] == 0:   
        Color[i] = 1  
        DFS(i)

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);

ПРИМЕР РЕАЛИЗАЦИИ НА ЯЗЫКЕ PYTHON

Color = [0] \* (n + 1)  
CycleFound = False  
  
def DFS(start):  
    Color[start] = 1  
    for u in V[start]:  
        if Color[u] == 0:  
            DFS(u)  
        elif Color[start] == 1:

            CycleFound = True   
    Color[start] = 2

for i in range(1, n + 1):

    if Color[i] == 0:

        DFS(i)

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());

ПРИМЕР РЕАЛИЗАЦИИ НА ЯЗЫКЕ PYTHON

Visited = [False] \* (n + 1)  
Ans = []

def DFS(start):  
    Visited[start] = True  
    for u in V[start]:  
        if not Visited[u]:  
            DFS(u)  
    Ans.append(start)

for i in range(1, n + 1):

    if not Visited(i):   
        DFS(i)   
Ans = Ans[::-1]

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



Алгоритм поиска в ширину

Информатика. Язык программирования Python в курсе информатики. Все классы.

**Алгоритм поиска в ширину** (англ. breadth-first search, BFS) позволяет найти кратчайшие пути из одной вершины невзвешенного (ориентированного или неориентированного) графа до всех остальных вершин. Под кратчайшим путем подразумевается путь, содержащий наименьшее число ребер.

Алгоритм построен на простой идее — пусть до какой-то вершины u найдено кратчайшее расстояние и оно равно d, а до вершины v кратчайшее расстояние не меньше, чем d. Тогда если вершины u и v – смежны, то кратчайшее расстояние до вершины v равно d+1.

Через d[i] будем обозначать кратчайшее расстояние до вершины i. Пусть начальная вершина имеет номер s, тогда d[s]=0. Для всех вершин смежных с s расстояние равно 1, для вершин, смежных с теми, до которых расстояние равно 1, расстояние равно 2 (если только оно не равно 0 или 1) и т. д.

Таким образом, организовать процесс вычисления кратчайших расстояний до вершин можно следующим образом. Для каждой вершины в массиве d будем хранить кратчайшее расстояние до этой вершины, если же расстояние неизвестно — будем хранить значение  -1 или None (в языке Python). В самом начале расстояние до всех вершин равно -1 (None), кроме начальной вершины, до которой расстояние равно 0. Затем перебираем все вершины, до которых расстояние равно 0, перебираем смежные с ними вершины и для них записываем расстояние равное 1. Затем перебираем все вершины, до которых расстояние равно 1, перебираем их соседей, записываем для них расстояние, равное 2 (если оно до этого было равно -1 (None)). Затем перебираем вершины, до которых расстояние было равно 2 и тем самым определяем вершины, до которых расстояние равно 3 и т. д. Этот цикл можно повторять либо пока обнаруживаются новые вершины на очередном шаге, либо n−1 раз (где n – число вершин в графе), так как длина кратчайшего пути в графе не может превосходить n−1.

Такая реализация алгоритма будет неэффективной, если на каждом шаге перебирать все вершины, отбирая те, которые были обнаружены на последнем шаге. Для эффективной реализации следует использовать очередь.

В очередь будут закладываться вершины после того, как до них будет определено кратчайшее расстояние. То есть очередь будет содержать вершины, которые были «обнаружены» алгоритмом, но не были рассмотрены исходящие ребра из этих вершин. Можно также сказать, что это очередь на «обработку» вершин.

Из очереди последовательно извлекаются вершины, рассматриваются все исходящие из них ребра. Если ребро ведет в не обнаруженную до этого вершину, то есть расстояние до новой вершины не определено, то оно устанавливается равным на единицу больше, чем расстояние до обрабатываемой вершины, а новая вершина добавляется в конец очереди.

Таким образом, если из очереди извлечена вершина с расстоянием d, то в конец очереди будут добавляться вершины с расстоянием d+1, то есть в любой момент исполнения алгоритма очередь состоит из вершин, удаленных на расстояние d, за которыми следуют вершины, удаленные на расстояние d+1.

Запишем алгоритм поиска в ширину.

РЕАЛИЗАЦИЯ НА ЯЗЫКЕ C++

vector<int> D(n + 1, -1);  
D[start] = 0;  
queue<int> Q;  
Q.push(s);  
while (!Q.empty())  
{  
    int u = Q.front();  
    Q.pop();  
    for (auto v : V[u])  
        if (D[v] == -1)  
        {  
            D[v] = D[u] + 1;  
            Q.push(v);  
        }  
}

РЕАЛИЗАЦИЯ НА ЯЗЫКЕ PYTHON

D = [None] \* (n + 1)  
D[start] = 0  
Q = [start]  
Qstart = 0  
while Qstart < len(Q):  
    u = Q[Qstart]  
    Qstart += 1   
    for v in V[u]:   
        if D[v] is None:   
            D[v] = D[u] + 1   
            Q.append(v)

В этом алгоритме n — число вершин в графе, пронумерованных от 1 до n. Номер начальной вершины (от которой ищутся пути) хранится в переменной start. Q — очередь, в которой хранятся обрабатываемые элементы (в примере на языке Python используйется список, Qstart — первый элемент очереди, добавление новой вершины в конец очереди — это вызов метода append для списка, удаление вершины из начала очереди — это увеличение Qstart на 1 (при этом первый элемент в очереди хранится в Q[Qstart])).

В самом начале в очередь добавляется только один элемент start, для которого в самом начале определено расстояние D[start] = 0 (для всех остальных элементов расстояние не определено). Цикл продолжается пока очередь не пуста (что в примере на Python проверяется условием Qstart < len(Q)). В цикле из очереди удаляется первый элемент u. Затем перебираются все смежные с ним вершины v. Если вершина v не была обнаружена ранее, что проверяется при помощи условия D[v] == -1 (D[v] is None в Python), то расстояние до вершины v устанавливается равным расстоянию до вершины u, увеличенному на 1, затем вершина v добавляется в конец очереди.

Если в графе содержится n вершин и m ребер, то сложность такого алгоритма равна O(n+m), так как алгоритму необходимо пройти по всем ребрам. Если граф хранится при помощи матрицы смежности, то сложность алгоритма равна O(n2), так как внутренний цикл перебора всех смежных вершин будет содержать n шагов для каждой обработанной вершины графа.

РЕАЛИЗАЦИЯ НА ЯЗЫКЕ PASCAL

var  
  Q, D: array[1..1001] of longint;  
  a: array[1..1000, 1..1000] of longint;  
  i, j, n, start, finish, Qstart, Qend, u, v: longint;  
begin  
  {считываем исходные данные:  
  количество вершин графа,  
  матрицу смежности,  
  номера стартовой и конечной вершины}  
  readln(n);  
  for i := 1 to n do  
    for j := 1 to n do  
      read(a[i, j]);  
  readln(start, finish);      
  {заполняем массив длин D}  
  for i := 1 to n do  
    D[i] := -1;  
  {расстояние от старта до старта равно нулю}    
  D[start] := 0;  
  {кладем стартовую вершину в очередь.  
  В очереди Q индекс Qstart - номер первого элемента очереди, Qend - номер ячейки после последнего элемента}  
  Q[1] := start;  
  Qstart := 1;  
  Qend := 2;  
  {Пока очередь не пуста, то есть, там есть хотя бы один элемент, то есть Qstart и Qend отличаются хотя бы на 1}  
  while Qstart < Qend do   
  begin  
    u := Q[Qstart];//забираем первую вершину из очереди  
    inc(Qstart); //передвигаем индекс первого элемента очереди  
    {перебираем все вершины графа}  
    for v := 1 to n do   
      {если нашли соседа, у которого расстояние еще не вычислено}  
      if (a[u, v] = 1) and (d[v] = -1) then begin{вычисляем его и кладем этого соседа в очередь}  
        d[v] := d[u] + 1;  
        Q[Qend] := v;  
        inc(Qend);  
      end;  
  end;  
  writeln(d[finish]);  
end.

Записанный алгоритм находит только кратчайшие расстояния до каждой из вершин графа. Чтобы найти кратчайший путь необходимо для каждой вершины хранить все ребра, по которым совершалось открытие новых вершин, то есть для каждой вершины необходимо хранить номер её предшественника — вершины, из которой была открыта данная вершина. Все сохраненные ребра вместе образуют дерево кратчайших путей. Чтобы построить кратчайший путь из начальной вершины до какой-то другой достижимой из нее вершины, необходимо взять путь в этом дереве, соединяющий эти две вершины.

РЕАЛИЗАЦИЯ НА ЯЗЫКЕ C++

Предшественников будем хранить в векторе:

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

Значение prev[i] есть номер предшествующей вершине i кратчайшего пути из вершины start. То есть чтобы построить кратчайший путь до вершины i необходимо построить кратчайший путь до вершины prev[i], а затем добавить к нему ребро из prev[i] в i.

При обнаружении новой вершины v в записанном алгоритме необходимо пометить, что данная вершина была достигнута проходом по ребру из вершины u, то есть предшественником вершины v является вершина u:

prev[v] = u;

Для восстановления ответа (кратчайшего пути от вершины start до некоторой вершины finish) заведем вектор ans для сохранения ответа, затем будет последовательно переходить от каждой вершины к ее предшественнику, пока не дойдем до значения -1, то есть отсутствия предшественника:

vector <int> ans;  
int curr = finish;  
while (curr != -1)  
{  
    ans.push\_back(curr);  
    curr = prev[curr];  
}

В итоге вектор ans будет хранить вершины на кратчайшем пути от start до finish, записанные в обратном порядке.

РЕАЛИЗАЦИЯ НА ЯЗЫКЕ PYTHON

Предшественников будем хранить в списке:

Prev = [None] \* (n + 1)

Значение Prev[i] есть номер предшествующей вершине i кратчайшего пути из вершины start. То есть чтобы построить кратчайший путь до вершины i необходимо построить кратчайший путь до вершины Prev[i], а затем добавить к нему ребро из Prev[i] в i.

При обнаружении новой вершины v в записанном алгоритме необходимо пометить, что данная вершина была достигнута проходом по ребру из вершины u, то есть предшественником вершины v является вершина u:

Prev[v] = u

Для восстановления ответа (кратчайшего пути от вершины start до некоторой вершины finish) заведем список Ans для сохранения ответа, затем будет последовательно переходить от каждой вершины к ее предшественнику, пока не дойдем до значения None, то есть отсутствия предшественника:

Ans = []  
curr = finish  
while curr is not None:  
    Ans.append(curr)  
    curr = Prev[curr]

В итоге список Ans будет хранить вершины на кратчайшем пути от start до finish, записанные в обратном порядке.

РЕАЛИЗАЦИЯ НА ЯЗЫКЕ PASCAL

…  
for i := 1 to n do   
  begin  
    D[i] := -1;  
    Prev[i] := -1;  
  end;    
…  
Prev[v] := u;  
…  
tmp := finish;  
  i := 1;  
  while tmp <> -1 do   
  begin  
    ans[i] := tmp;  
    inc(i);  
    tmp := Prev[tmp];  
  end;  
  for j := i - 1 downto 1 do

    write(ans[j], ' ');



Алгоритм Дейкстры

Информатика. Язык программирования Python в курсе информатики. Все классы.

Алгоритм Дейкстры

**Алгоритм Дейкстры** назван в честь голландского ученого Эдсгера Дейкстры (Edsger Dijkstra). Алгоритм был предложен в 1959 году для нахождения кратчайших путей от одной вершины до всех остальных в ориентированном взвешенном графе, при условии, что все ребра в графе имеют неотрицательные веса.

Рассмотрим две модели хранения взвешенного графа в памяти. В первой модели (матрица весов, аналог матрицы смежности) будем считать, что вес ребра из вершины i в вершину j равен w[i][j], то есть в матрице w хранятся веса ребра для любых двух вершин. Если из вершины i в вершину j нет ребра, то w[i][j]==INF для некоторого специального значения константы INF. Значение INF следует выбирать исходя из задачи, например, если речь идет о расстояниях между какими-либо населенными пунктами Земли, то можно выбрать значение INF равным 109 километров.

Алгоритм Дейкстры относится к так называемым «жадным» алгоритмам. Пусть расстояние от начальной вершины start до вершины i хранится в массиве dist[i]. Начальные значения dist[start]=0, dist[i]=INF для всех остальных вершин i. То есть в самом начале алгоритму известен путь из вершины start до вершины start длины 0, а до остальных вершин кратчайшие пути неизвестны. Между тем алгоритм будет постепенно улучшать значения в массиве dist, в результате получит кратчайшие расстояния до всех вершин.

Основная идея для улучшения называется «релаксацией ребра». Пусть из вершины i в вершину j есть ребро веса w[i][j], при этом выполнено неравенство dist[i] + w[i][j] < dist[j]. То есть можно построить маршрут из начальной вершины до вершины i и добавить к нему ребро из i в j, и суммарная стоимость такого маршрута будет меньше, чем известная ранее стоимость маршрута из начальной вершины в вершину j. Тогда можно улучшить значение dist[j], присвоив dist[j] = dist[i] + w[i][j].

В алгоритме Дейкстры вершины красятся в два цвета, будем говорить, что вершина «неокрашенная» или «окрашенная». Изначально все вершины неокрашенные. Если алгоритм Дейкстры покрасил вершину i, то это означает, что найденное значение dist[i] является наилучшим возможным и в последствии не будет улучшаться, то есть значение dist[i] является кратчайшим расстоянием от начальной вершины до вершины i. Если же вершина не покрашена, то величина dist[i] для такой вершины i равна кратчайшему пути из вершины start до вершины i, который проходит только по покрашенным вершинам (за исключением самой вершины i).

На каждом шаге алгоритма Дейкстры красится одна новая вершина. В качестве такой вершины выбирается неокрашенная вершина i с наименьшим значением D[i]. Затем рассматриваются все ребра, исходящие из вершины i, и производится релаксация этих ребер, то есть улучшаются расстояния до вершин, смежных с i.

Алгоритм заканчивается, когда на очередном шаге не останется неокрашенных вершин или если расстояние до всех неокрашенных вершин будет равно INF (то есть эти вершины являются недостижимыми).

Запишем алгоритм Дейкстры. Пусть n — число вершин в графе, вершины пронумерованы от 0 до n−1. Номер начальной вершины — start и веса ребер хранятся в матрице w.

РЕАЛИЗАЦИЯ НА ЯЗЫКЕ C++

const int INF = 1000000000;  
vector <int> dist(n, INF);  
dist[start] = 0;  
vector <bool> used(n);  
int min\_dist = 0;  
int min\_vertex = start;  
while (min\_dist < INF)  
{  
    int i = min\_vertex;  
    used[i] = true;  
    for (int j = 0; j < n; ++j)  
        if (dist[i] + w[i][j] < dist[j])  
                dist[j] = dist[i] + w[i][j];  
    min\_dist = INF;  
    for (int j = 0; j < n; ++j)  
        if (!used[j] && dist[j] < min\_dist)  
        {  
            min\_dist = dist[j];  
            min\_vertex = j;  
        }  
}

РЕАЛИЗАЦИЯ НА ЯЗЫКЕ PYTHON

INF = 10 \*\* 10  
dist = [INF] \* n  
dist[start] = 0  
used = [False] \* n  
min\_dist = 0  
min\_vertex = start  
while min\_dist < INF:  
    i = min\_vertex   
    used[i] = True   
    for j in range(n):   
        if dist[i] + w[i][j] < dist[j]:   
            dist[j] = dist[i] + w[i][j]   
    min\_dist = INF  
    for j in range(n):  
        if not used[j] and dist[j] < min\_dist:  
            min\_dist = dist[j]  
            min\_vertex = j

Массив used будет хранить информацию о том, была ли покрашена вершина. Сначала инициализируются массивы dist и used. Затем запускается внешний цикл алгоритма, который выбирает неокрашенную вершину с минимальным расстоянием, номер этой вершины хранится в переменной min\_vertex, а расстояние до этой вершины — в переменной min\_dist. Если же min\_dist оказывается равно INF, то значит все неокрашенные вершины являются недостижимыми и алгоритм заканчивает свою работу. Иначе найденная вершина окрашивается и после этого релаксируются все ребра, исходящие из этой вершины.

Данный алгоритм имеет сложность O(n2), так как внешний цикл может быть выполнен до n раз, внутри него содержится два цикла, каждый из которых также выполняется n раз.

Для восстановления ответа, то есть для нахождения пути из начальной вершины до всех остальных, необходимо построить дерево кратчайших путей. Это дерево будет состоять из тех ребер, которые были успешно срелаксированы в результате исполнения алгоритма. То есть если происходит релаксация ребра из i в j, то теперь кратчайший маршрут из вершины start до вершины j должен проходить через вершину i и затем содержать ребро i-j. Тем самым вершина i становится предшественником вершины j на кратчайшем пути из начальной вершины до вершины j.

Рассмотрим реализацию алгоритм Дейкстры с восстановлением ответа на графе, хранимым в виде списка смежности на языке C++. Ребро из вершины i в вершину j веса wt будет хранить в виде пары (j, wt), список ребер, исходящих из вершины i будет храниться в векторе w[i]. То есть списки смежности w будут объявлены так:

vector <vector <pair <int, int > > > w;

Реализация считывания ребер графа (из номеров вершин будем вычитать число 1 для нумерации с нуля, рассматриваем ориентированный граф, то есть не дублируем ребро):

for (k = 0; k < m; ++k)  
{  
    int i, j, wt;  
    cin >> i >> j >> wt;  
    w[i - 1].push\_back(make\_pair(j - 1, wt));  
}

Тогда при обработки вершины i вместо перебора всех других вершин мы рассматриваем только ребра, исходящие из данной вершины.

const int INF = 1000000000;  
vector <int> dist(n, INF);  
dist[start] = 0;  
vector <int> prev(n, -1);  
vector <bool> used(n);  
int min\_dist = 0;  
int min\_vertex = start;  
while (min\_dist < INF)  
{  
    int i = min\_vertex;  
    used[i] = true;  
    for (auto edge: w[i])  
    {  
        int j = edge.first;  
        int wt = edge.second;  
        if (dist[i] + wt < dist[j])  
        {  
            dist[j] = dist[i] + wt;  
            prev[j] = i;  
        }  
    }  
    min\_dist = INF;  
    for (int j = 0; j < n; ++j)  
        if (!used[j] && dist[j] < min\_dist)  
        {  
            min\_dist = dist[j];  
            min\_vertex = j;  
        }  
}

Восстановление ответа производится аналогично поиску в ширину или в глубину.

vector <int> path;  
while (j != -1)  
{  
    path.push\_back(j);  
    j = prev[j];  
}  
reverse(path.begin(), path.end());

Рассмотрим реализацию алгоритм Дейкстры с восстановлением ответа на графе, хранимым в виде списка смежности на языке Python. Набор вершин, смежных с вершиной i будет храниться в множестве w[i]. Также необходимо хранить веса ребер, будем считать, что для хранения весов ребер используется словарь weight, где ключом является кортеж из двух вершин. То есть вес ребра из i в j хранится в элементе weight[i, j] словаря весов.

dist = [INF] \* n  
dist[start] = 0  
prev = [None] \* n  
used = [False] \* n  
min\_dist = 0  
min\_vertex = start  
while min\_dist < INF:  
    i = min\_vertex  
    used[i] = True  
    for j in w[i]:  
        if dist[i] + weight[i, j] < dist[j]:  
            dist[j] = dist[i] + weight[i, j]   
            prev[j] = i  
    min\_dist = INF  
    for i in range(n):   
    if not used[i] and dist[i] < min\_dist:   
        min\_dist = dist[i]   
        min\_vertex = i

Для нахождения кратчайшего пути из вершины start до вершины j будем переходить от каждой вершины к ее предшественнику:

path = []  
while j is not None:  
    path.append(j)   
    j = prev[j]   
path = path[::-1]

Алгоритм Дейкстры применим только в том случае, когда веса всех ребер неотрицательные. Это гарантирует то, что после окраски расстояние до вершины не может быть улучшено. Если в графе могут быть ребра отрицательного веса, то следует использовать другие алгоритмы.

Реализация алгоритма Дейкстры с использованием кучи или контейнера set в STL

Алгоритм Дейкстры в ранее приведенной реализации имеет сложность O(n2). В этой реализации производится выбор элемента с наименьшим расстоянием до него, что производится путем просмотра всех вершин. Если хранить все неокрашенные вершины в куче или в контейнере set STL (которое реализовано при помощи сбалансированного дерева поиска), то поиск очередной вершины для окрашивания можно производить более оптимально.

Но обновление расстояния до другой  в этом случае будет выполняться за O(log⁡n), так как это требует перестройки кучи или дерева поиска. Если в графе m ребер, то максимальное число релаксаций ребер также будет не больше m и суммарная сложность всех релаксаций будет O(mlog⁡n). Таким образом, алгоритм Дейкстры с использованием кучи будет иметь сложность O(nlog⁡n+mlog⁡n)=O((n+m)log⁡n). Если граф — разреженный, то такой алгоритм работает существенно быстрее, чем обычный алгоритм Дейкстры, но на плотных графах (если m∼n2) он, наоборот, менее эффективен, чем простая реализация Дейкстры.

Приведем реализацию алгоритма Дейкстры с использованием структуры set на языке STL. Нам необходимо извлекать из set элемент с наименьшим расстоянием до него, а также узнавать при этом номер извлеченной вершины. Для этого в структуре set мы будем хранить пары значений (dist[i], i), то есть пары, у который первый компонент - расстояние до вершины, второй компонент - номер вершины. Поскольку пары сортируются лексикографически по полю first, а при равенстве - по полю second, то объекты в нашем set будут упорядочены прежде всего по возрастанию значения dist и в начале set будет храниться вершина с минимальным значением set.

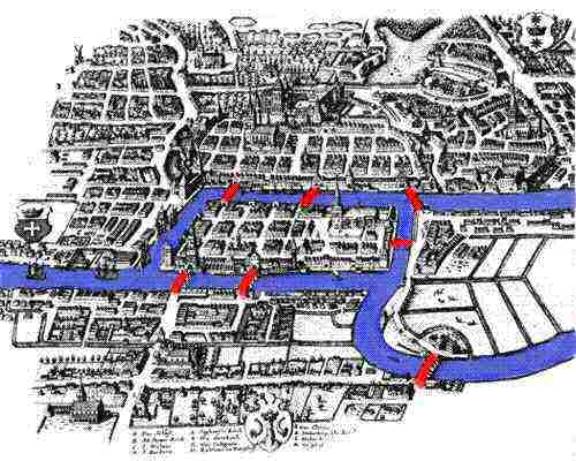
vector<int> dist(n, INF);  
dist[start] = 0;  
set<pair<int, int> > unused;  
unused.insert(make\_pair(0, start));  
while (!unused.empty())  
{  
    int i = unused.begin()->second;  
    unused.erase(unused.begin());  
    for (auto edge : w[i])  
    {  
        int j = edge.first;  
        int wt = edge.second;  
        if (dist[i] + wt < dist[j])  
        {  
            unused.erase(make\_pair(dist[j], j));   
            dist[j] = dist[i] + wt;  
            unused.insert(make\_pair(dist[j], j));  
        }  
    }  
}

При этом в set мы будем хранить только вершины, которые достижимы из начальной, то есть для них значение dist меньше 0. Поэтому в самом начале в set кладется только одна начальная вершина. Внутри главного цикла из set удаляется начальный элемент, а дальше при обработке вершины рассматриваются все исходящие из нее ребра и для каждого ребра соответствующая ему пара удаляется из set (если вершина не была ранее достижима, т.е. до нее расстояние было равно INF, то операция удаления ничего не сделает, но и не приведет к ошибке), а потом в set добавляется новая пара, соответствующая измененному значению расстояния до данной вершины.

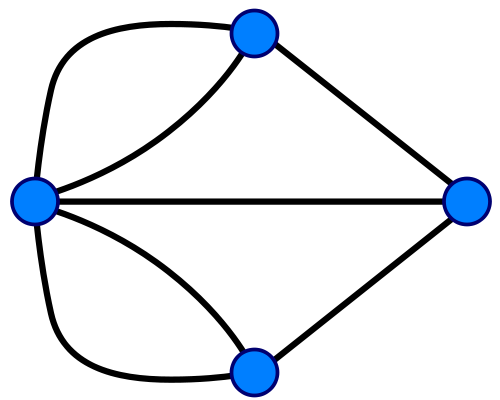
Эйлеровы графы

Информатика. Язык программирования Python в курсе информатики. Все классы.

**Задача о кенигсбергских мостах.** Город Кенигсберг (ныне Калининград) расположен на берегах реки Прегель и двух островах на этой реке. Части города соединены мостами. Спрашивается, можно ли, выйдя из какой-нибудь точки города, пройти по каждому мосту ровно один раз и вернуться в исходную точку.



Объектами в данной задаче являются части города, а связями — мосты. Поэтому этой задаче соответствует следующий граф:



Задача о кенигсбергских мостах эквивалентна вопросу о том, можно ли «обойти» данный граф, пройдя по каждому ребру ровно один раз и вернуться в исходную вершину, то есть существует ли в графе простой цикл, проходящий по всем ребрам.

ОПРЕДЕЛЕНИЕ

**Эйлеровым путем** называется простой путь, содержащий все ребра. **Эйлеровым циклом**называется простой цикл, содержащий все ребра. **Эйлеровым графом** называется граф, в котором есть эйлеров цикл.

ТЕОРЕМА ЭЙЛЕРА

Граф без изолированных вершин является эйлеровым тогда и только тогда, когда он связен и степени всех его вершин четны.

Гамильтоновы графы

Информатика. Язык программирования Python в курсе информатики. Все классы.

ОПРЕДЕЛЕНИЕ

**Гамильтоновым путем**называется путь, проходящий через каждую вершину ровно один раз. **Гамильтоновым циклом**называется цикл, проходящий через каждую вершину ровно один раз. **Гамильтоновым графом** называется граф, в котором есть гамильтонов цикл.

ТЕОРЕМА ОРЕ

Пусть дан обыкновенный связный граф, содержащий n>2 вершин, такой что сумма степеней любых двух несмежных вершин не меньше, чем n. Тогда граф гамильтонов.

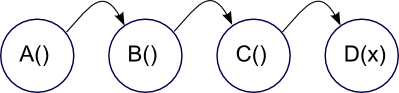


Граф вызовов функций

Информатика. Язык программирования Python в курсе информатики. Все классы.

Граф вызовов функции отображает связь функций друг с другом по логике их вызовов.

Заметим, что он может быть составлен не только для функций, но и для процедур, то есть для любых подпрограмм. В языках Си и Python процедуры синтаксически не отличаются от функций и даже не выделяются в отдельное понятие. Т.е. все подпрограммы — функции, только некоторые могут ничего не возвращать.

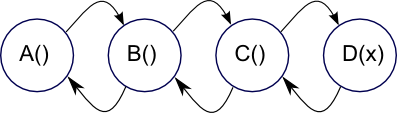


По данному графу видно, что функция A непосредственно вызывает функцию B, которая, в свою очередь, вызывает функцию C(), а та пользуется функцией D().

После вызова функцией А функции B, она сама погружается в ожидание завершения подпрограммы.

Таким образом, функции A() важен результат B(), и она не может без него продолжить свое выполнение.

Если мы отобразим на графе обратный ход, как возвращается исполнение к функциям вместе с результатами вычислений с более глубоких уровней вызова, то получится вот так:



ПРИМЕР НА ЯЗЫКЕ СИ

//заголовки функций:  
int A();  
int B();  
int C();  
int D();  
  
//тела функций  
int A()  
{  
    printf("A() started\n");  
    int result = B() + 1;  
    printf("A() finished\n");  
    return result;  
}  
  
int B()  
{  
    printf("B() started\n");  
    int result = C() + 1;  
    printf("B() finished\n");  
    return result;  
}  
  
int C()  
{  
    printf("C() started\n");  
    int result = D() + 1;  
    printf("C() finished\n");  
    return result;  
}  
  
int D()  
{  
    printf("D() started\n");  
    int result = 1; //никого не вызывает - это последняя по глубине вызовов функция  
    printf("D() finished\n");  
    return result;  
}

ПРИМЕР НА ЯЗЫКЕ PYTHON

def A():  
    print('A() started')  
    result = B() + 1  
    print('A() finished')  
    return result  
   
def B():  
    print('B() started')  
    result = C() + 1  
    print('B() finished')  
    return result  
   
def C():  
    print('C() started')  
    result = D() + 1  
    print('C() finished')  
    return result  
   
def D():  
    print('D() started')  
    result = 1 # никого не вызывает - это последняя по глубине вызовов функция  
    print('D() finished')  
    return result

ЧТО ВЕРНЕТ ФУНКЦИЯ A() В МЕСТО СВОЕГО ВЫЗОВА?

4

**Это значение будет зависеть от значений, возвращенных остальными функциями.**

В КАКОМ ПОРЯДКЕ БУДУТ НАПЕЧАТАНЫ X() STARTED И X() FINISHED ДЛЯ A, B, C, D?

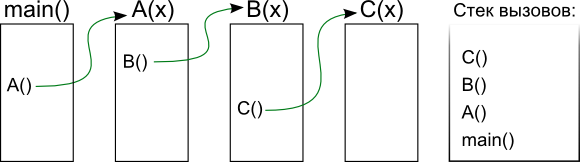
A() started  
B() started  
C() started  
D() started  
D() finished  
C() finished  
B() finished  
A() finished

Причина обратного порядка завершения в том, что возврат из функций происходит именно в обратном порядке. Пока не завершится D, функция C простаивает и ожидает от нее результат. Так же и другие функции.



Стек вызовов

Информатика. Язык программирования Python в курсе информатики. Все классы.



**Стек вызовов** (call stack) — стек, хранящий информацию для возврата управления из подпрограмм (функций) в программу или подпрограмму (при вложенных или рекурсивных вызовах).

При вызове подпрограммы в стек заносится **адрес возврата** — адрес в памяти следующей инструкции приостанавливаемой программы, а управление передается подпрограмме. При последующем вложенном или рекурсивном вызове в стек заносится очередной адрес возврата и так далее.

При возврате из подпрограммы адрес возврата снимается со стека, и управление передается на следующую инструкцию приостановленной программы (или подпрограмм