* [Теория графов](https://foxford.ru/lessons/32275/conspects/1)
* [Алгоритм поиска в ширину](https://foxford.ru/lessons/32275/conspects/2)

**Теория графов**

**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.

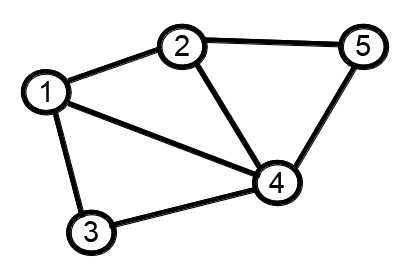
**Способы представления графов в памяти**

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]]**

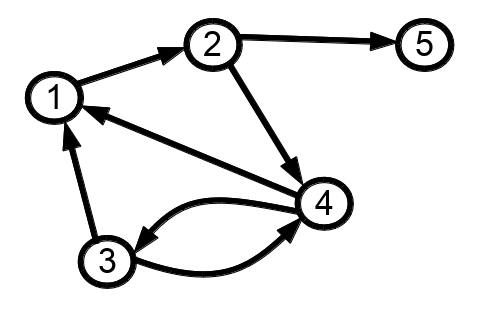
Поскольку нумерация в нашем примере начинается с 0, то к списку добавлен еще один фиктивный элемент W[0].

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.

#### Алгоритм поиска в ширину

**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], ' ');**