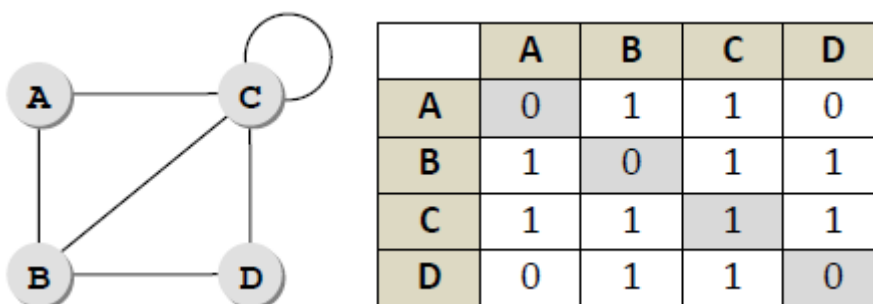


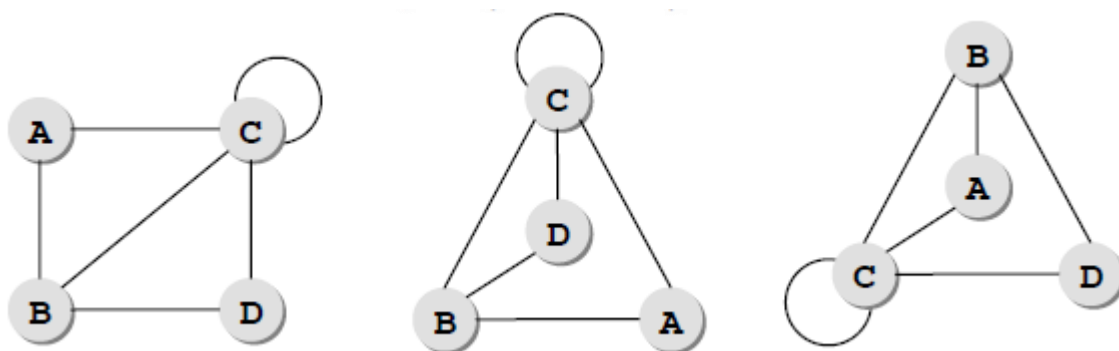
Графы

Граф – это набор вершин (узлов) и связей между ними (рёбер). Информацию о вершинах и рёбрах графа обычно хранят в виде таблицы специального вида – матрицы смежности:



Единица на пересечении строки A и столбца B означает, что между вершинами A и B есть связь. Ноль указывает на то, что связи нет. Матрица смежности симметрична относительно главной диагонали (серые клетки в таблице). Единица на главной диагонали обозначает *петлю* – ребро, которое начинается и заканчивается в одной и той же вершине (в данном случае – в вершине C).

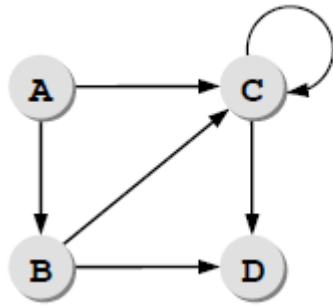
Строго говоря, граф – это математический объект, а не рисунок. Конечно, его можно нарисовать на плоскости, но матрица смежности не даёт никакой информации о том, как именно следует располагать вершины друг относительно друга. Для таблицы, приведенной выше, возможны, например, такие варианты:



В рассмотренном примере все узлы связаны, то есть, между любой парой вершин существует *путь* – последовательность рёбер, по которым можно перейти из одной вершины в другую. Такой граф называется *связным*.

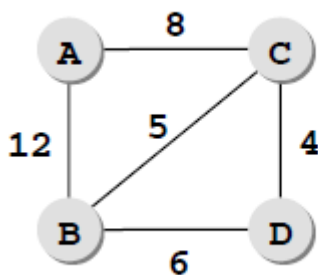
Вспоминая материал предыдущего пункта, можно сделать вывод, что *дерево* – это частный случай связного графа, в котором нет замкнутых путей – *циклов*.

Если для каждого ребра указано направление, граф называют *ориентированным* (или *орграфом*). Рёбра орграфа называют *дугами*. Его матрица смежности не всегда симметричная. Единица, стоящая на пересечении строки A и столбца B говорит о том, что существует дуга из вершины A в вершину B



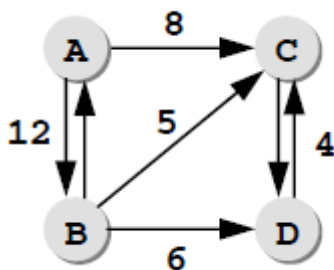
	A	B	C	D
A	0	1	1	0
B	0	0	1	1
C	0	0	1	1
D	0	0	0	0

Часто с каждым ребром связывают некоторое число – *вес ребра*. Это может быть, например, расстояние между городами или стоимость проезда. Такой граф называется взвешенным. Информация о взвешенном графе хранится в виде весовой матрицы, содержащей веса рёбер:



	A	B	C	D
A		12	8	
B	12		5	6
C	8	5		4
D		6	4	

У взвешенного орграфа весовая матрица может быть несимметрична относительно главной диагонали:

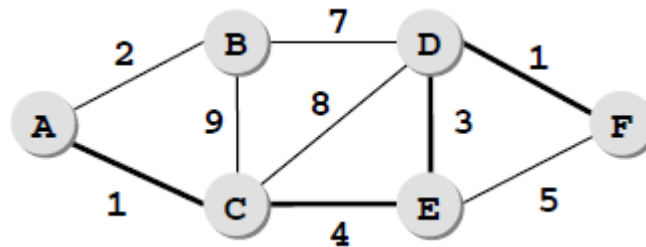


	A	B	C	D
A		12	8	
B	12		5	6
C				4
D			4	

Если связи между двумя вершинами нет, на бумаге можно оставить ячейку таблицы пустой, а при хранении в памяти компьютера записывать в нее условный код, например, 0, -1 или очень большое число (∞), в зависимости от задачи.

«Жадные» алгоритмы

Задача. Известна схема дорог между несколькими городами. Числа на схеме (рисунок ниже) обозначают расстояния (дороги не прямые, поэтому неравенство треугольника может нарушаться):

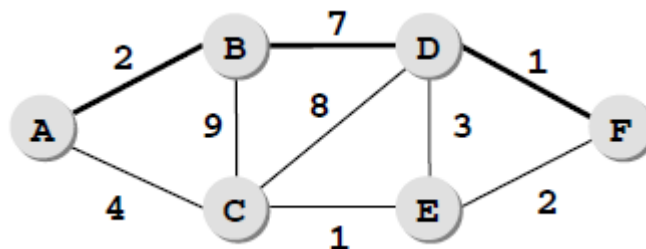


*Нужно найти кратчайший маршрут из города **A** в город **F**.*

Первая мысль, которая приходит в голову – на каждом шаге выбирать кратчайший маршрут до ближайшего города, в котором мы еще не были. Для заданной схемы на первом этапе едем в город C (длина 1), далее – в E (длина 4), затем в D (длина 3) и наконец в F (длина 1). Общая длина маршрута равна 9.

Алгоритм, который мы применили, называется «жадным». Он состоит в том, чтобы на каждом шаге многоходового процесса выбирать наилучший в данный момент вариант, не думая о том, что впоследствии этот выбор может привести к худшему решению.

Для данной схемы жадный алгоритм на самом деле дает оптимальное решение, но так будет далеко не всегда. Например, для той же задачи с другой схемой:



жадный алгоритм даст маршрут A-B-D-F длиной 10, хотя существует более короткий маршрут A-C-E-F длиной 7.

Жадный алгоритм не всегда позволяет получить оптимальное решение.

Однако есть задачи, в которых жадный алгоритм всегда приводит к правильному решению. Одна из таких задач (её называют задачей Прима-Крускала в честь Р. Прима и Д. Крускала, которые независимо предложили её в середине XX века) формулируется так:

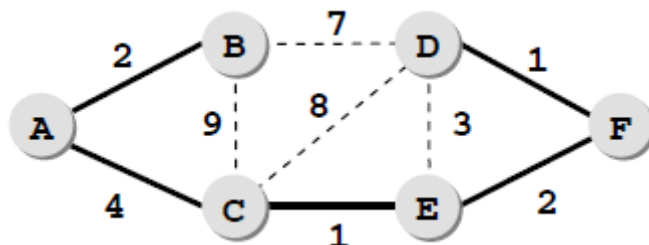
Задача. В стране Лимонии есть **N** городов, которые нужно соединить линиями связи. Между какими городами нужно проложить линии связи, чтобы все города были связаны в одну систему, и общая длина линий связи была наименьшей?

В теории графов эта задача называется задачей построения минимального остовного дерева (то есть дерева, связывающего все вершины). Остовное дерево для связного графа с **N** вершинами имеет **N-1** ребро.

Рассмотрим жадный алгоритм решения этой задачи, предложенный Крускалом:

2. на каждом шаге к будущему дереву добавляется ребро минимального веса, которое ещё не было выбрано и не приводит к появлению цикла.

На рисунке показано минимальное остовное дерево для одного из рассмотренных выше графов



Здесь возможна такая последовательность добавления рёбер: CE, DF, AB, EF, AC. Обратите внимание, что после добавления ребра EF следующее «свободное» ребро минимального веса – это DE (длина 3), но оно образует цикл с рёбрами DF и EF, и поэтому не было включено в дерево.

При программировании этого алгоритма сразу возникает вопрос: как определить, что ребро еще не включено в дерево и не образует цикла в нём? Существует очень красивое решение этой проблемы, основанное на раскраске вершин.

Сначала все вершины раскрашиваются в разные цвета, то есть все рёбра из графа удаляются. Таким образом, мы получаем множество элементарных деревьев (так называемый лес), каждое из которых состоит из одной вершины. Затем последовательно соединяем отдельные деревья, каждый раз выбирая ребро минимальной длины, соединяющее разные деревья (выкрашенные в разные цвета). Объединённое дерево перекрашивается в один цвет, совпадающий с цветом одного из вошедших в него поддеревьев. В конце концов все вершины оказываются выкрашены в один цвет, то есть все они принадлежат одному остовному дереву. Можно доказать, что это дерево будет иметь минимальный вес, если на каждом шаге выбирать подходящее ребро минимальной длины.

В программе сначала присвоим всем вершинам разные числовые коды:

```
col = [i for i in range(N)]
```

Здесь **N** – количество вершин, а **col** – «цвета» вершин (список из N элементов).

Затем в цикле **N-1** раз (именно столько рёбер нужно включить в дерево) выполняем следующие операции:

1. ищем ребро минимальной длины среди всех рёбер, концы которых окрашены в разные цвета;

2. найденное ребро в виде кортежа(**iMin**, **jMin**) добавляется в список выбранных, и все вершины, имеющие цвет **col[jMin]**, перекрашиваются в цвет **col[iMin]**.

Приведем полностью основной цикл программы:

```
ostov = []
for k in range(N-1):
    # поиск ребра с минимальным весом
    minDist = 1e10          # очень большое число
    for i in range(N):
        for j in range(N):
            if col[i] != col[j] and W[i][j] < minDist:
                iMin = i
                jMin = j
                minDist = W[i][j]
            # добавление ребра в список выбранных
            ostov.append( (iMin, jMin) )
            # перекрашивание вершин
            c = col[jMin]
            for i in range(N):
                if col[i] == c:
                    col[i] = col[iMin]
```

Здесь **W** – весовая матрица размера N на N (индексы строк и столбцов начинаются с 0); **ostov** – список хранения выбранных рёбер (для каждого ребра хранится кортеж из номеров двух вершин, которые оно соединяет). Если связи между вершинами *i* и *j* нет, в элементе **W[i][j]** матрицы будем хранить «бесконечность» – число, намного большее, чем длина любого ребра. При этом начальное значение переменной **minDist** должно быть ещё больше.

После окончания цикла остается вывести результат – рёбра из массива **ostov**

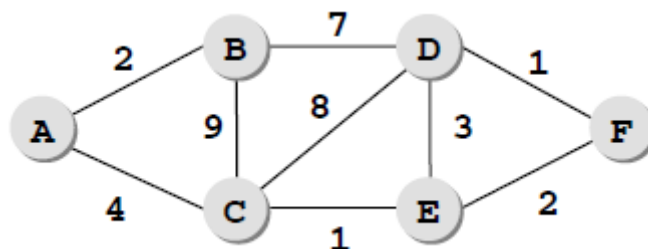
```
for edge in ostov:
    print ( "(", edge[0], ",", edge[1], ")" )
```

В цикле перебираются все элементы списка **ostov**; каждый из них – кортеж из двух элементов – попадает в переменную **edge**, так что номера вершин определяются как **edge[0]** и **edge[1]**.

Кратчайшие маршруты

На примере задачи выбора кратчайшего маршрута мы увидели, что в ней жадный алгоритм не всегда дает правильное решение. В 1960 году Э. Дейкстра предложил алгоритм, позволяющий найти все кратчайшие расстояния от одной вершины графа до всех остальных и соответствующие им маршруты. Предполагается, что длины всех рёбер (расстояния между вершинами) положительные.

Рассмотрим уже знакомую схему, в которой не сработал жадный алгоритм:



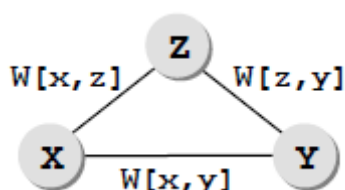
Алгоритм Дейкстры использует дополнительные массива: в одном (назовем его R) хранятся кратчайшие (на данный момент) расстояния от исходной вершины до каждой из вершин графа, а во втором (массив P) – вершина, из которой нужно приехать в данную вершину.

Сначала записываем в массив R длины рёбер от исходной вершины A до всех вершин, а в соответствующие элементы массива P – вершину A:

	A	B	C	D	E	F
R	0	2	4	∞	∞	∞
P	x	A	A			

Знак ∞ , обозначает, что прямого пути нет из вершины A в данную вершину нет (в программе вместо ∞ можно использовать очень большое число). Таким образом, вершина A уже рассмотрена и выделена серым фоном. В первый элемент массива P записан символ x, обозначающий начальную точку маршрута (в программе можно использовать несуществующий номер вершины, например, «-1» или **None**).

Из оставшихся вершин находим вершину с минимальным значением в массиве R: это вершина B. Теперь проверяем пути, проходящие через эту вершину: не позволят ли они сократить маршрут к другим, которые мы ещё не посещали. Идея состоит в следующем: если сумма весов $W[x, z] + W[z, y]$ меньше, чем вес $W[x, y]$, то из вершины X лучше ехать в вершину Y не напрямую, а через вершину Z:



Проверяем наш граф: ехать из A в C через B невыгодно (получается путь длиной 11 вместо 4), а вот в вершину D можно проехать (путь длиной 4), поэтому запоминаем это значение вместо ∞ в массиве R, и записываем вершину B на соответствующее место в массив P («в D приезжаем из B»):

	A	B	C	D	E	F
R	0	2	4	9	∞	∞
P	x	A	A	B		

Вершины E и F по-прежнему недоступны.

Следующей рассматриваем вершину C (для нее значение в массиве R минимально). Оказывается, что через неё можно добраться до E (длина пути 5):

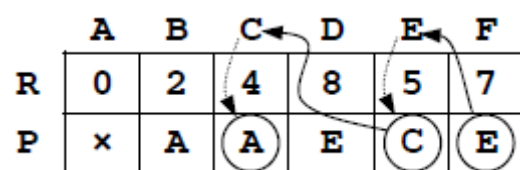
	A	B	C	D	E	F
R	0	2	4	9	5	∞
P	x	A	A	B	C	

Затем посещаем вершину E, которая позволяет достигнуть вершины F и улучшить минимальную длину пути до вершины D:

	A	B	C	D	E	F
R	0	2	4	8	5	7
P	x	A	A	E	C	E

После рассмотрения вершин F и D таблица не меняется. Итак, мы получили, что кратчайший маршрут из A в F имеет длину 7, причем он приходит в вершину F из E. Как же получить весь маршрут? Нужно просто посмотреть в массиве P, откуда лучше всего ехать в E – выясняется, что из вершины C, а в вершину C – напрямую из начальной точки A

	A	B	C	D	E	F
R	0	2	4	8	5	7
P	x	A	A	E	C	E



Поэтому кратчайший маршрут **A-C-E-F**. Обратите внимание, что этот маршрут «раскручивается» в обратную сторону, от конечной вершины к начальной. Заметим, что полученная таблица содержит все кратчайшие маршруты из вершины A во все остальные, а не только из A в F.

Алгоритм Дейкстры можно рассматривать как своеобразный «жадный» алгоритм: действительно, на каждом шаге из всех невыбранных вершин выбирается вершина X, длина пути до которой от вершины A минимальна. Однако можно доказать, что это расстояние – действительно минимальная длина пути от A до X. Предположим, что для всех предыдущих выбранных вершин это свойство справедливо. При этом X – это ближайшая не выбранная вершина, которую можно достичь из начальной точки, проезжая только через выбранные вершины. Все остальные пути в X, проходящие через ещё не выбранные вершины, будут длиннее, поскольку все рёбра имеют положительную длину. Таким образом, найденная длина пути из A в X – минимальная. После завершения алгоритма, когда все вершины выбраны, в массиве R находятся длины кратчайших маршрутов.

Теперь напишем программу на Python. Переменная N будет обозначать количество вершин графа, «список списков» W – это весовая матрица, её удобно вводить из файла. Логический массив `active` хранит состояние вершин (просмотрена или не просмотрена): если значение `active[i]` истинно, то вершина активна (ещё не просматривалась).

В начале программы присваиваем начальные значения:

```
active = [True]*N # все вершины не просмотрены
R = W[0][:]      # скопировать в R строку 0 весовой матрицы
P = [0]*N        # только прямые маршруты из вершины 0
```

Обратите внимание, что нельзя написать

```
R = W[0] # неверное копирование строки W[0] в массив R!
```

потому что таким образом мы записываем в переменную R ссылку на строку 0 матрицы W , и при любом изменении массива R нулевая строка матрицы W также будет изменяться. Добавление «`[:]`» создаёт срез этого массива «от начала до конца», и в переменную R записывается ссылка на новый объект, который будет независим от матрицы W .

Сразу помечаем, что вершина 1 просмотрена (не активна), с нее начинается маршрут.

```
active[0] = False
P[0] = -1
```

В основном цикле, который выполняется $N-1$ раз (так, чтобы все вершины были просмотрены) среди активных вершин ищем вершину с минимальным соответствующим значением в массиве R и проверяем, не лучше ли ехать через неё:

```
for i in range(N-1):
    # поиск новой рабочей вершины R[j] -> min
    minDist = 1e10 # очень большое число
    for j in range(N):
        if active[j] and R[j] < minDist:
            minDist = R[j]
            kMin = j
    active[kMin] = False
    # проверка маршрутов через вершину kMin
    for j in range(N):
        if R[kMin] + W[kMin][j] < R[j]:
            R[j] = R[kMin] + W[kMin][j]
            P[j] = kMin
```

В конце программы выводим оптимальный маршрут (здесь – до вершины с номером $N-1$) в обратном порядке следования вершин:


```

i = N-1
while i >= 0: # для начальной вершины P[i]=-1
    print ( i, end = " " )
    i = P[i]   # переход к следующей вершине

```

Теперь рассмотрим более общую задачу: найти все кратчайшие маршруты из любой вершины во все остальные. Как мы видели, алгоритм Дейкстры находит все кратчайшие пути только из одной заданной вершины. Конечно, можно было бы применить этот алгоритм N раз, но существует более красивый метод – *алгоритма Флойда-Уоршелла*, основанный на той же самой идее сокращения маршрута: иногда бывает короче ехать через промежуточные вершины, чем напрямую. На языке Python этот алгоритм записывается так:

```

for k in range(N):
    for i in range(N):
        for j in range(N):
            if W[i][k] + W[k][j] < W[i][j]:
                W[i][j] = W[i][k] + W[k][j]

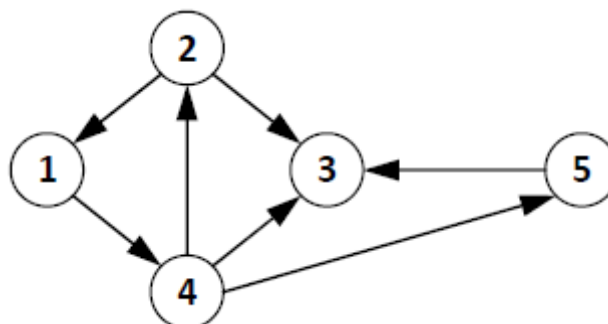
```

В результате исходная весовая матрица графа W размером N на N превращается в матрицу, хранящую длины оптимальных маршрутов. Для того, чтобы найти сами маршруты, нужно использовать еще одну дополнительную матрицу, которая выполняет ту же роль, что и массив P в алгоритме Дейкстры. Подробное описание и реализацию этого алгоритма вы можете найти в литературе или в Интернете.

Использование списков смежности

Граф можно описать с помощью списков смежности. Поскольку в Python есть встроенный тип данных «список», при программировании на этом языке такое описание очень удобно использовать.

Список смежности для какой-то вершины – это множество вершин, с которыми связана данная вершина. Рассмотрим орграф, состоящий из 5 вершин:



Для него списки смежности (с учетом направлений рёбер) выглядят так:

вершина 1: (4)

вершина 2: (1,3)

вершина 3: ()

вершина 4: (2,3, 5)

вершина 5: (3)

Граф, показанный на рисунке выше, описывается в виде вложенного списка так:

```
Graph = [ [],          # фиктивный элемент
          [4],         # список смежности для вершины 1
          [1,3],       # ... для вершины 2
          [],          # ... для вершины 3
          [2,3,5],     # ... для вершины 4
          [3] ]        # ... для вершины 5
```

Для того, чтобы использовать нумерацию вершин с 1, мы добавили фиктивный элемент – пустой список смежности для несуществующей вершины 0.

Используя такое представление, построим функцию **pathCount**, которая находит количество путей из одной вершины в другую. Алгоритм её работы основан на следующей идее: общее количество путей из вершины **X** в вершину **Y** равно сумме количеств путей из **X** в **Y** через все остальные вершины. Чтобы избежать циклов (замкнутых путей), при этом нужно учитывать только те вершины, которые ещё не посещались. Эту информацию необходимо где-то запоминать, для этой цели мы, будем использовать список посещённых вершин.

Таким образом, в функцию **pathCount** нужно передать следующие данные (аргументы):

- описание графа в виде списков смежности для каждой вершины, **graph**;
- номера начальной и конечной вершин, **vStart** и **vEnd**;
- список посещённых вершин, **visited**.

Тогда основной цикл функции **pathCount** приобретает вид

```
count = 0
for v in graph[vStart]:
    if not v in visited:
        count += pathCount ( graph, v, vEnd, visited )
```

Вы, конечно, заметили, что функция **pathCount** получилась рекурсивной, то есть она вызывает сама себя (в цикле). Поэтому нужно определить условие окончания рекурсии: если начальная и конечная вершины совпадают, то существует только один путь, и можно сразу выйти из функции с помощью оператора **return**:

```
if vStart == vEnd:
    return 1
```

Приведём полный текст функции

```
def pathCount ( graph, vStart, vEnd, visited = None ):
    if vStart == vEnd: return 1
    if visited is None: visited = []
```

```
visited.append( vStart )
count = 0
for v in graph[vStart]:
    if not v in visited:
        count += pathCount( graph, v, vEnd, visited )
visited.pop()
return count
```

и основную программу, которая находит количество путей из вершины 1 в вершину 3:

```
Graph = [ [], [4], [1,3], [], [2,3,5], [3] ]
print ( pathCount (Graph, 1, 3) )
```

У этой программы есть два существенных недостатка. Во-первых, она не выводит сами маршруты, а только определяет их количество. Во-вторых, некоторые данные могут вычисляться повторно: если мы уже нашли количество путей из какой-то вершины X в конечную вершину, то когда это значение потребуется снова, желательно сразу использовать полученный ранее результат, а не вызывать функцию рекурсивно ещё раз. Попробуйте улучшить программу самостоятельно так, чтобы исправить эти недостатки.