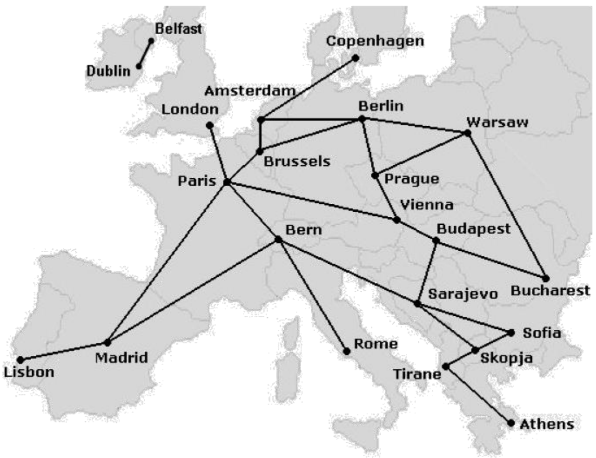
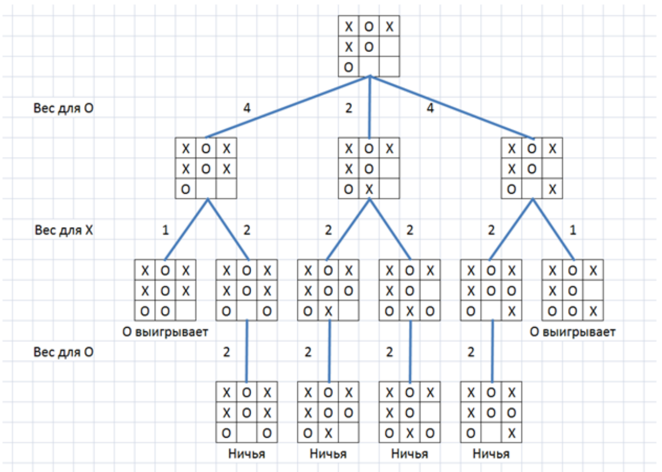
{ ЛЕКЦИЯ 4 }

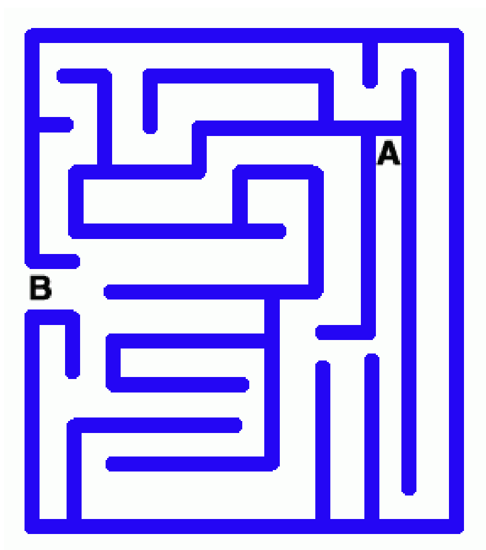
{ Алгоритмы на графах }

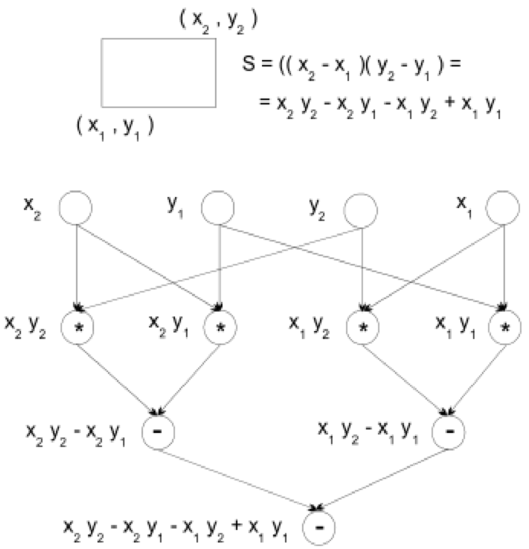
{ Графы }

Задача маршрутизации (задача коммивояжёра). Анализ игр. Анализ алгоритмов. Задача о максимальном потоке.









{ Графы }

Граф - абстрактный математический объект, представляющий собой множество вершин и рёбер.

Граф без петель и кратных рёбер называется **простым**.

Степень вершины – количество инцидентных (присоединённых) ей рёбер.

Подграф – это граф в котором множество вершин и рёбер являются подмножествами соответствующих множеств исходного графа.

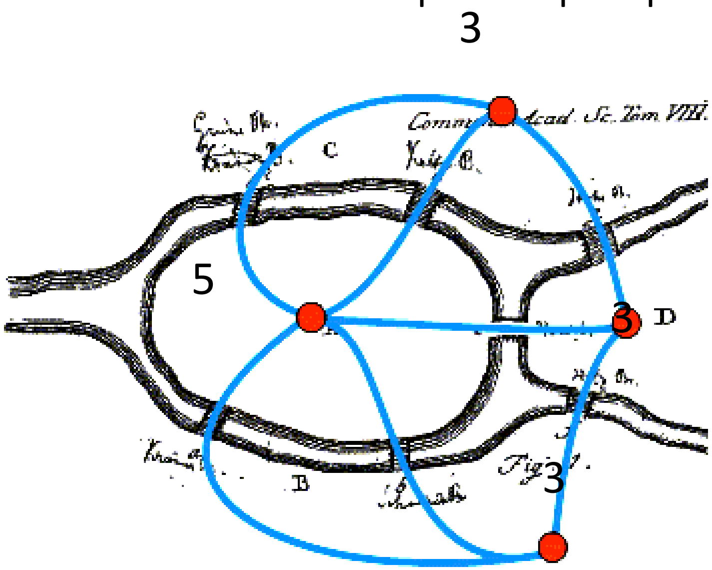
Цикл Эйлера – неповторяющийся обход всех рёбер.

Задача о кёнигсбергских мостах не имеет решения, т.к.:

5-2 + 3-2 + 3-2 + 3-2 – больше 2-х нечётных вершин.

Маршрут – последовательность вершин и рёбер.

Компонента связности - максимальный (по включению) связный подграф.



Задача о семи кёнигсбергских мостах: обойти все мосты пройдя по каждому один раз
(1736 г., Леонард Эйлер)

Дерево - это связный ациклический граф (число вершин = число рёбер + 1).

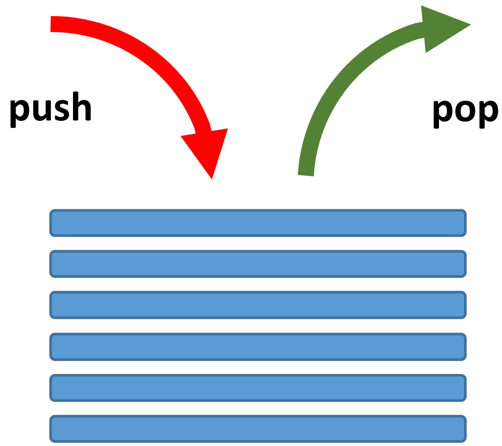
Корневое дерево – дерево в котором одна из вершин – корень.

Остовное дерево (остов) — это подграф данного графа, содержащий все его вершины и являющийся деревом.

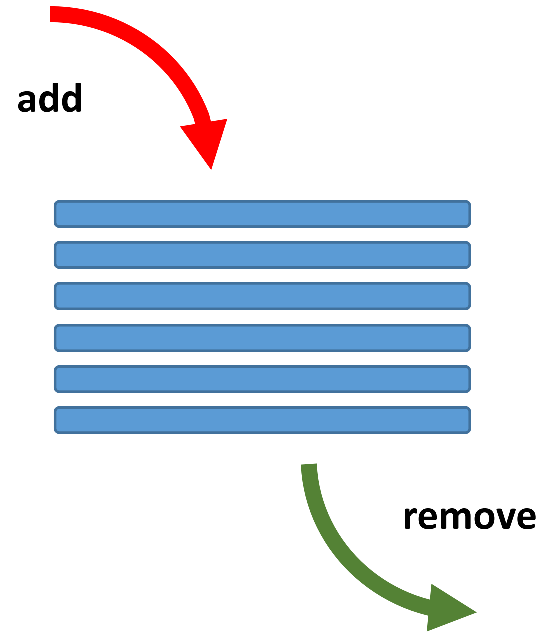
Взвешенной граф – граф в котором каждому ребру приписано значение (вес).

{ Очередь и стек. }

Stack (стек)
LIFO (Last in, first out)
Последним пришел, первым вышел



Queue (очередь)
FIFO (First in, first out)
Первым пришел, первым вышел



Основные операции со **стеком** это:

- *Push* – вставка элемента наверх стека;
- *Top* – получение верхнего элемента без удаления;
- *Pop* – получение верхнего элемента и его удаление;
- Проверка пуст ли стек.

Основные операции с **очередью** это:

- добавить элемент в конец очереди;
- получить элемент из начала очереди (без удаления из очереди);
- получить элемент из начала очереди (с удалением из очереди);
- проверка пустая ли очередь.

В ***Python*** стеком можно сделать любой список, так как для них доступны операции *pop* и *push*.

{ Очередь и стек в *Python*. }

```
queue = list(range(1,22,3))
print(queue)
```

```
[1, 4, 7, 10, 13, 16, 19]
```

```
%%time
x = queue.pop(0)
```

```
Wall time: 547 ns
```

```
print(x, queue)
```

```
1 [4, 7, 10, 13, 16 , 19]
```

Не эффективно по **времени**

```
queue = list(range(1,22,3))
print(queue)
i=0
```

```
[1, 4, 7, 10, 13, 16, 19]
```

```
%%time
x = queue[i]
i+=1
```

```
Wall time: 103 ns
```

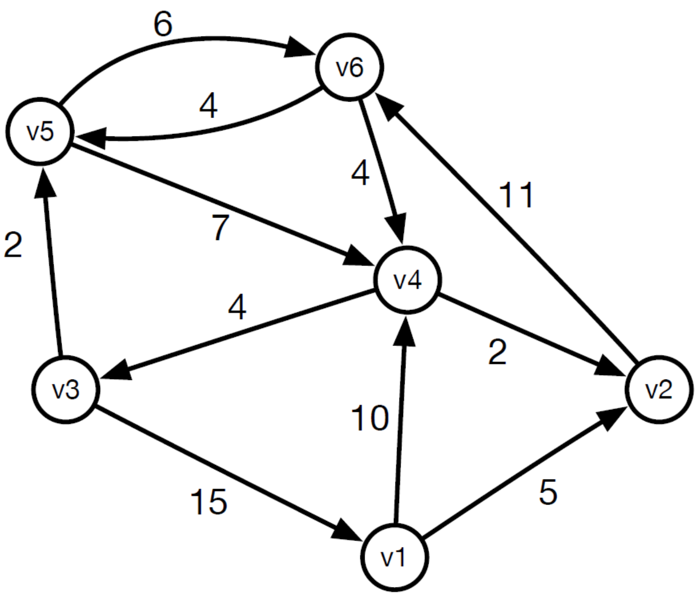
```
print(x, queue)
```

```
1 [1, 4, 7, 10, 13, 16 , 19]
```

Не эффективно по **памяти**

{ Представление графов }

Граф может быть представлен:



Графически

	1	2	3	4	5	6
1	0	5	0	10	0	0
2	0	0	0	0	0	11
3	15	0	0	0	2	0
4	0	2	4	0	0	0
5	0	0	0	7	0	6
6	0	0	0	4	4	0

Матрицей смежности

Множеством смежности $v_1:\{2,4\},\, v_2:\{6\},\, v_3:\{1,5\},\, v_4:\{2,3\},\, v_5:\{3,4,6\},\, v_6:\{4,5\}.$

Списком рёбер {откуда, куда, стоимость}: $\{1,2,5\},\{1,4,10\},\{2,6,11\},\{3,1,15\},\{3,5,2\},$
$\{4,2,2\},\{5,4,7\},\{5,6,6\},\{6,4,4\},\{6,5,4\}.$

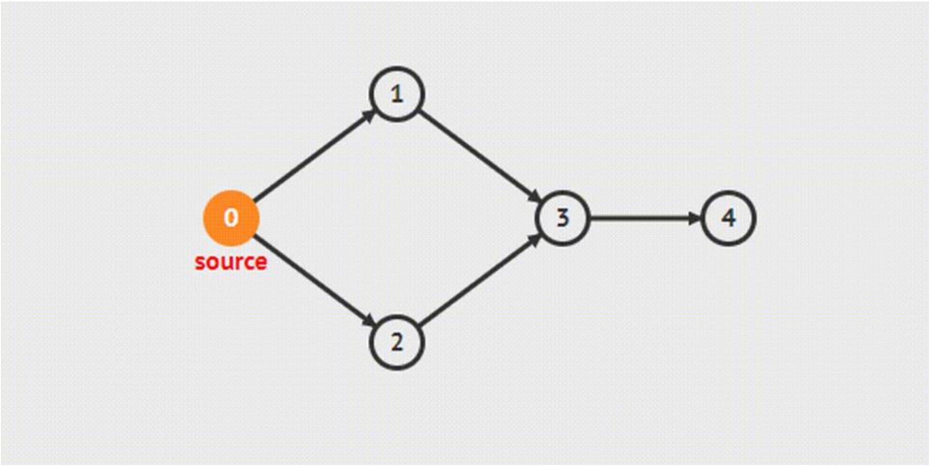
{ Обход графа }

Во многих приложениях нужно уметь выписывать все вершины графа по одному разу, начиная с некоторой. Это делается с помощью обходов в глубину или в ширину.

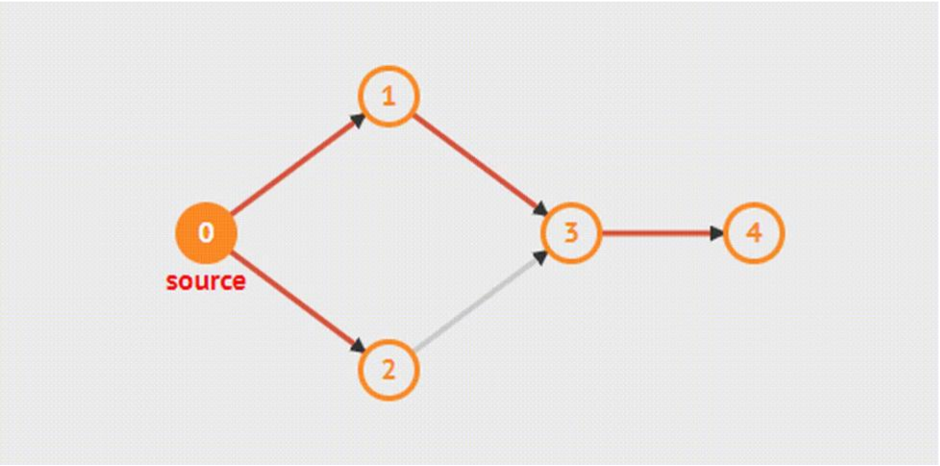
Основная идея обходов:

- на каждом шаге рассмотреть очередную необработанную вершину;
- пометить эту вершину некоторым образом;
- до/после обработки данной вершины осуществить обход из всех нерассмотренных соседей.

Для упорядочивания вершин используется очередь (обход в ширину) или стек (обход в глубину).



Поиск в глубину



Поиск в ширину

{ Что даёт поиск }

Поиск в глубину можно использовать для:

- выделение компонент связности (подсчёт компонент);
- поиск простого цикла.

Поиск в ширину можно использовать для:

- выделение компонент связности (подсчёт компонент);
- поиск кратчайшего пути в невзвешенном графе;
- восстановление кратчайшего пути;
- нахождение кратчайшего цикла в ориентированном невзвешенном графе.

{ Поиск в ширину }

```
def bfs(adj, s, level):
	level[s] = 0                      # уровень начальной вершины
	queue = [s]                       # добавляем начальную вершину в очередь
	while queue:                      # пока там что-то есть
		v = queue.pop(0)              # извлекаем вершину
		for w in adj[v]:              # запускаем обход из вершины v
			if level[w] is -1:        # проверка на посещенность
				queue.append(w)       # добавление вершины в очередь
				level[w] = level[v] + 1   # подсчитываем уровень вершины
				print(level[w], level, queue)
```

```
adj = [          #список смежности
	[1,2],       # 0
	[3,4],       # 1
	[1,4],       # 2
	[4],         # 3
	[1,3,5],     # 4
	[0,2] ]      # 5
level = [-1] * len(adj)  #список уровней вершин
```

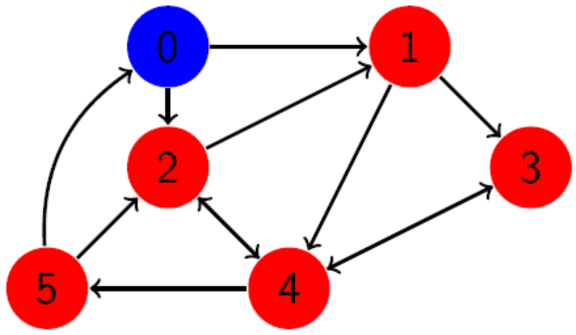
{ Поиск в ширину }

```
bfs(adj, 0, level)
```

```
1 [0, 1, -1, -1, -1, -1] [1]
1 [0, 1, 1, -1, -1, -1] [1, 2]
2 [0, 1, 1, 2, -1, -1] [2, 3]
2 [0, 1, 1, 2, 2, -1] [2, 3, 4]
3 [0, 1, 1, 2, 2, 3] [5]
```

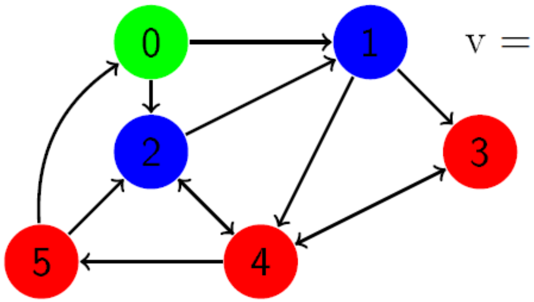
```
print(level)
```

```
[0, 1, 1, 2, 2, 3]
```

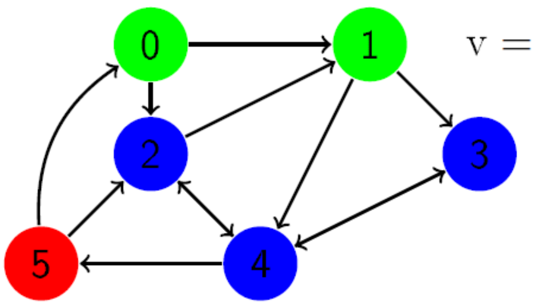


$q = [0]$

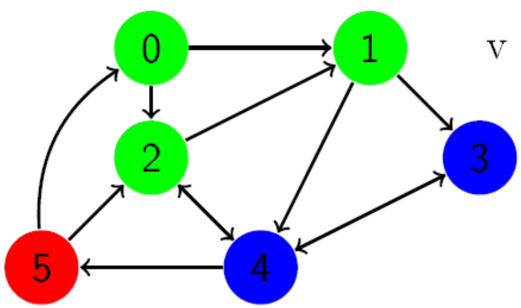
{ Поиск в ширину }



$$v = 0, d = [0, 1, 1, \infty, \infty, \infty], q = [1, 2]$$

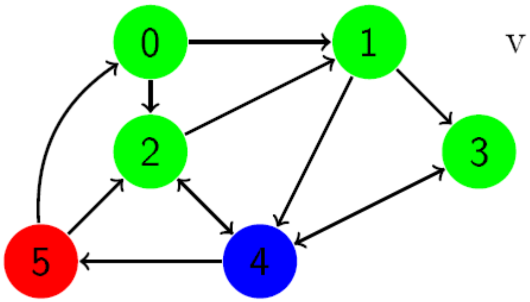


$$v = 1, d = [0, 1, 1, 2, 2, \infty], q = [2, 3, 4]$$

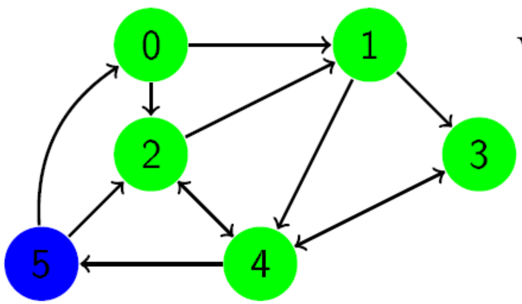


$$v = 2, d = [0, 1, 1, 2, 2, \infty], q = [3, 4]$$

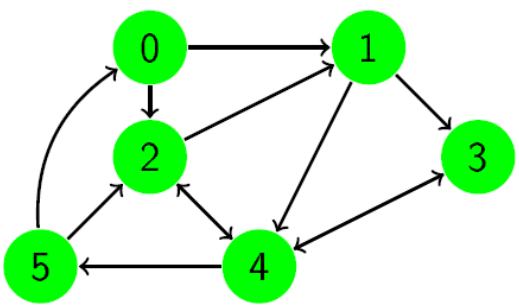
{ Поиск в ширину }



$$v = 3, d = [0, 1, 1, 2, 2, \infty], q = [4]$$



$$v = 4, d = [0, 1, 1, 2, 2, 3], q = [5]$$



$$v = 5, d = [0, 1, 1, 2, 2, 3], q = []$$

{ Поиск в глубину }

```
def dfs(adj, s, level):
	level[s] = 0                                  # уровень начальной вершины
	queue = [s]                                   # добавляем начальную вершину в очередь
	while queue:                                  # пока там что-то есть
		v = queue.pop(-1)                         # извлекаем вершину
		for w in adj[v]:                          # запускаем обход из вершины v
			if level[w] is -1:                    # проверка на посещенность
				queue.append(w)                   # добавление вершины в очередь
				level[w] = level[v] + 1           # подсчитываем уровень вершины
				print(level[w], level, queue)
```

```
adj = [          #список смежности
	[1,2],       # 0
	[3,4],       # 1
	[1,4],       # 2
	[4],         # 3
	[1,3,5],     # 4
	[0,2] ]      # 5
level = [-1] * len(adj)  #список уровней вершин
```

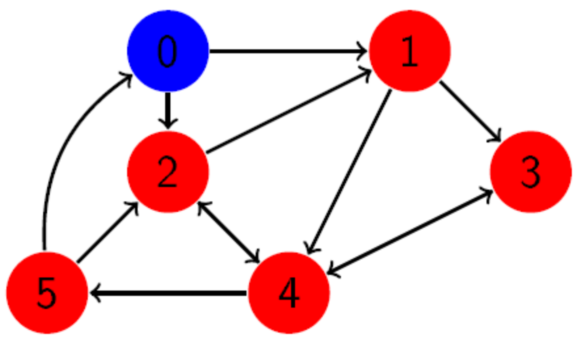
{ Поиск в глубину }

```
dfs(adj, 0, level)
```

```
1 [0, 1, -1, -1, -1, -1] [1]
1 [0, 1, 1, -1, -1, -1] [1, 2]
2 [0, 1, 1, -1, 2, -1] [1, 4]
3 [0, 1, 1, 3, 2, -1] [1, 3]
3 [0, 1, 1, 3, 2, 3] [1, 3, 5]
```

```
print(level)
```

```
[0, 1, 1, 3, 2, 3]
```



$q = [0]$

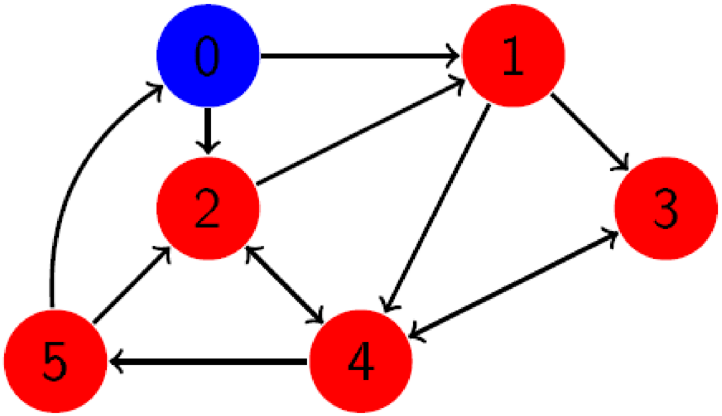
{ Восстановление кратчайшего пути }

```
def bfs_path(adj, s, level, parents):
    level[s] = 0                        # уровень начальной вершины
    queue = [s]                         # добавляем начальную вершину в очередь
    while queue:                        # пока там что-то есть
        v = queue.pop(0)                # извлекаем вершину
        for w in adj[v]:                # запускаем обход из вершины v
            if level[w] is -1:          # проверка на посещенность
                queue.append(w)         # добавление вершины в очередь
                parents[w] = v          # запоминаем предшественника
                level[w] = level[v] + 1 # подсчитываем уровень вершины
                print(level[w], level, queue)
```

```
adj = [             #список смежности
    [1,2],          # 0
    [3,4],          # 1
    [1,4],          # 2
    [4],            # 3
    [1,3,5],        # 4
    [0,2] ]         # 5
level = [-1] * len(adj)         #список уровней вершин
parents = [-1] * len(adj)       #список предшественников
```

{ Восстановление кратчайшего пути }

```
def PATH (end, parents):
    path = [end]
    parent = parents[end]
    while not parent is -1:
        path.append(parent)
        parent = parents[parent]
    return path[::-1]
```




```
bfs_path(adj, 0, level , parents)
```

```
print(level)
```

```
[0, 1, 1, 2, 2, 3]
```

```
print(parents)
```

```
[-1, 0, 0, 1, 1, 4]
```

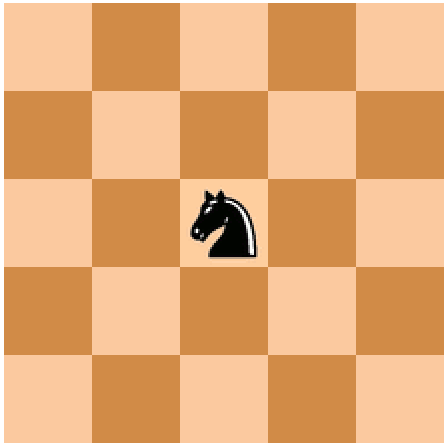
```
PATH (5, parents)
```

```
[0, 1, 4, 5]
```

{ Ход конём }

По каким клеткам должен пройти конь перемещаясь между клетками d4 и f2?

Сведем задачу к графу, пройдем его в ширину и восстановим кратчайший путь.



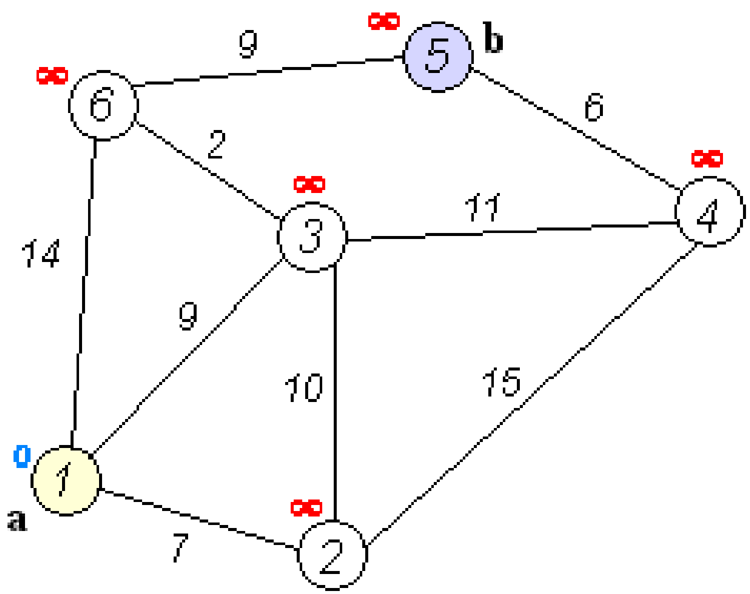
{ Алгоритм Дейкстры }

Часто нужно уметь вычислять расстояние от некоторой вершины графа до всех остальных:

- поиск кратчайшего географического пути.
- поиск гипотезы наименьшего веса (наибольшей вероятности) в задачах вычислительной лингвистики.
- поиск слова наименьшего веса, принимаемого конечным автоматом и т.д.

В случае рёбер стоимости 1 это делает алгоритм поиска в ширину, используя очередь.

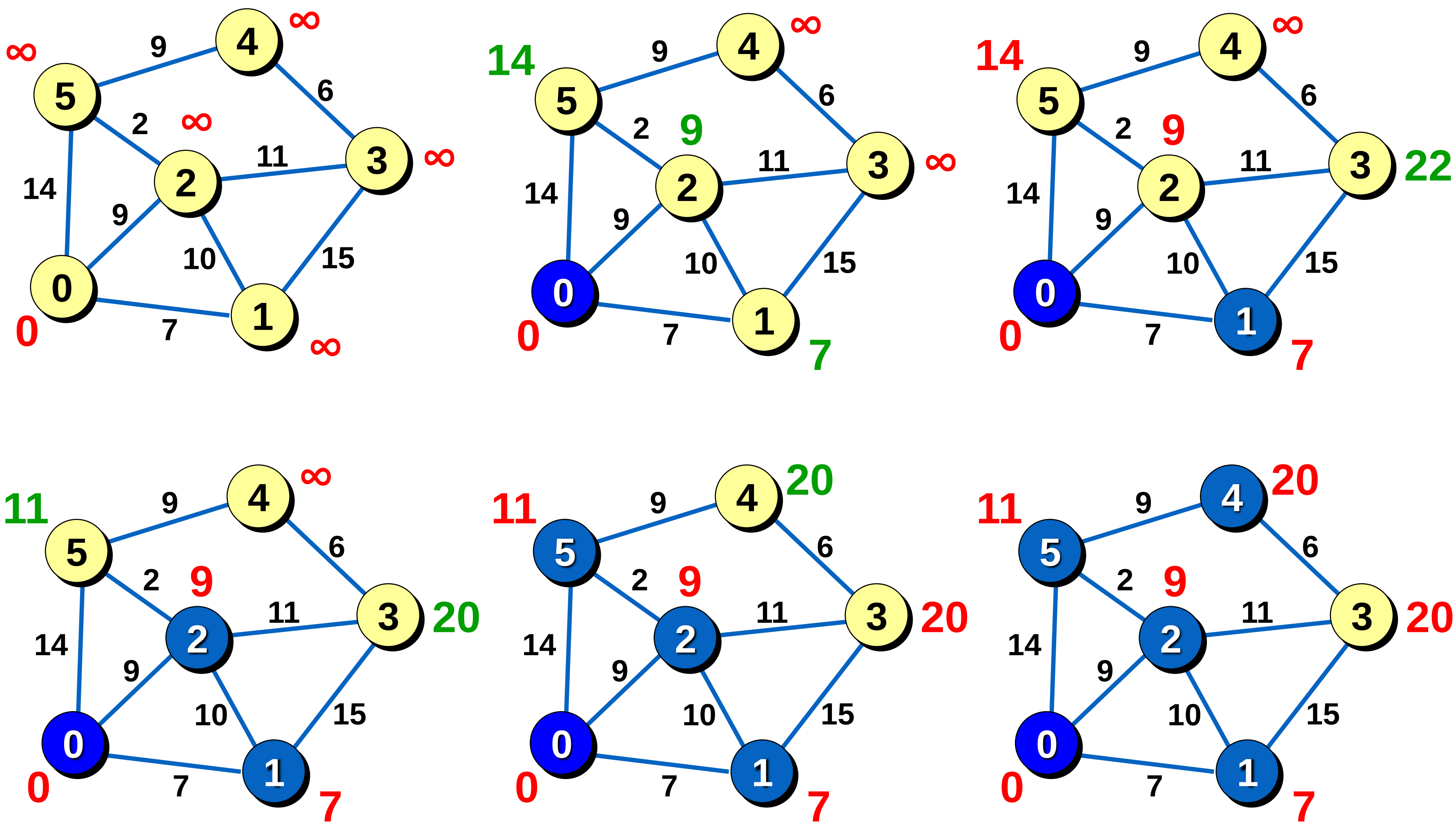
В случае рёбер произвольной длины это делает алгоритм Дейкстры.



Алгоритм:

1) присвоить всем вершинам метку ∞;
2) среди нерассмотренных вершин найти вершину j с наименьшей меткой;
3) для каждой необработанной вершины i: если путь к вершине i через вершину j меньше существующей метки, заменить метку на новое расстояние;
4) если остались необработанны вершины, перейти к шагу 2;
5) метка = минимальное расстояние.

{ Алгоритм Дейкстры }



{ Алгоритм Дейкстры }

Массивы:

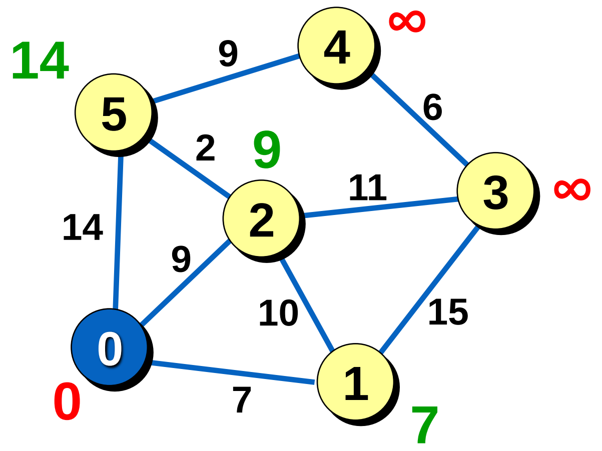
1) массив **a**, такой что **a[i]=1**, если вершина уже рассмотрена, и **a[i]=0**, если нет.
2) массив **b**, такой что **b[i]** – длина текущего кратчайшего пути из заданной вершины **x** в вершину **i**;
3) массив **c**, такой что **c[i]** – номер вершины, из которой нужно идти в вершину **i** в текущем кратчайшем пути.

Инициализация:

1) заполнить массив **a** нулями (вершины не обработаны);
2) записать в **b[i]** значение **W[x][i]**;
3) заполнить массив **c** значением **x**;
4) записать **a[x]=1**.

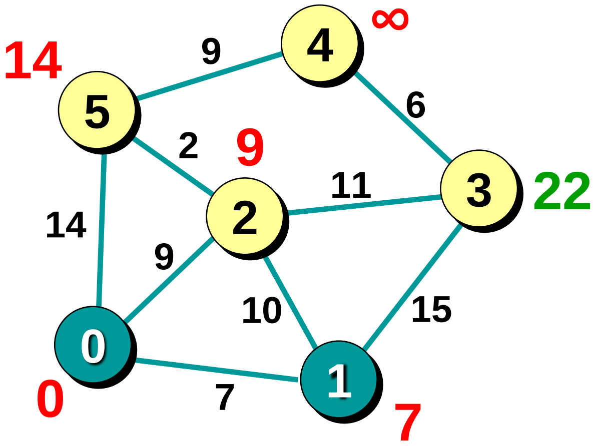
Основной цикл:

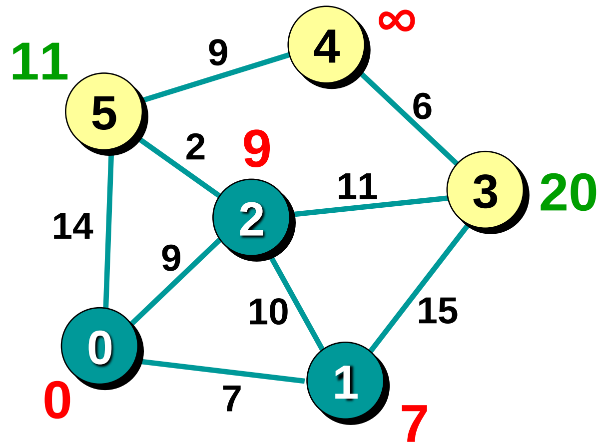
1) если все вершины рассмотрены, то стоп.
2) среди всех нерассмотренных вершин (**a[i]=0**) найти вершину **j**, для которой **b[i]** – минимальное;
3) записать **a[j]=1**;
4) для всех вершин **k**: если путь в вершину **k** через вершину **j** короче, чем найденный ранее кратчайший путь, запомнить его: записать **b[k]=b[j]+W[j][k]** и **c[k]=j**.

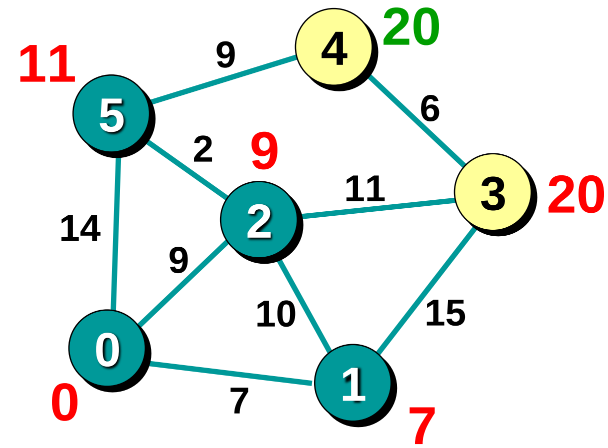


	0	1	2	3	4	5
a	1	0	0	0	0	0
b	0	7	9	∞	∞	14
c	0	0	0	0	0	0

{ Алгоритм Дейкстры }







	0	1	2	3	4	5
a	1	1	0	0	0	0
b	0	7	9	22	∞	14
c	0	0	0	1	0	0

	0	1	2	3	4	5
a	1	1	1	0	0	0
b	0	7	9	20	∞	11
c	0	0	0	2	0	2

	0	1	2	3	4	5
a	1	1	1	0	0	1
b	0	7	9	20	20	11
c	0	0	0	2	5	2

Маршрут из вершины 0 в вершину 4:

4 → 5 → 2 → 0

{ ВСЁ }